5SSD0

Bayesian Machine Learning and Information Processing

Probabilistic Programming

Bert de Vries (Flux 7.101)

Wouter Kouw (Flux 7.060)

5SSD0 Software Installation

- Please download the course materials from Github by pressing the green `Code` button and selecting `Download ZIP`. If you're familiar with git, we recommend cloning the repository.
- You will need certain pieces of software (VS Code, Julia, and Jupyter) to execute the lecture notebooks and work on the programming assignments. The instructions below will help you install them.

For Windows Users

- We shall use `juliaup`, a program that manages versions of the Julia kernel. You can download `juliaup` from the Windows store.
- After installation, go to your Start menu, type `cmd` and open the Command Prompt. Now run `juliaup add lts` and `juliaup default lts`. This adds the long-term stable version of the Julia kernel and sets it as the default. If you open Julia now (see Start menu), it should report `Version 1.10.5`.
- Download VS Code and follow the installation procedure.
- Open VS Code and press the Extensions button in the left toolbar (the four cubes symbol; `Ctrl+Shift+X`). Search for and install the following three extensions: `Python`, `Julia`, and `Jupyter`.
- In VS Code, go to `File` (top toolbar), and press `Open Folder` (`Ctrl+K+O`). Navigate to and select the `BMLIP` folder that you downloaded from Github. This will be your development environment for the course.
- To test for a succesful software installation, open one of the lesson notebooks (in the folder `lessons/notebooks/`) and select kernel on the top right. It should display the option `Julia` and then `julia lts channel`. You can then press the `run all` button on the top toolbar.

For Linux Users

- For Julia, we shall use `juliaup`, a program that manages versions of the Julia kernel. Open a terminal and execute

  ```
  curl -fsSL https://install.julialang.org | sh
  ```

  After installing `juliaup`, enter the following commands in the terminal to add the `lts` (long-term stable) version of the Julia kernel.

  ```
  juliaup add lts
   juliaup default lts
  ```

  To check which version of Julia you're running, execute `juliaup status` in a terminal. It should have an asterisk in front of the `lts` version.
- Next, go to the VS Code website, and download the installer for your Linux distro and follow the installation procedure.
- Open VS Code and press the `Extensions` button in the left toolbar (the four cubes symbol; `Ctrl+Shift+X`). Search for and install the following three extensions: `Python`, `Julia` and `Jupyter`.
- In VS Code, go to `File` (top toolbar), and press `Open Folder` (`Ctrl+K+O`). Navigate to and select the `BMLIP` folder that you downloaded from Github. This will be your development environment for the course.
- To test for a succesful software installation, open one of the lesson notebooks (in the folder `lessons/notebooks/`) and then press `Select kernel` on the top right. It should display the option `Julia`, and then `julia lts channel`. Finalize the test with `run all` (top toolbar).

For Mac Users

This installation guide assumes that you have homebrew installed. If not, please install it first.

- For Julia, we shall use `juliaup`, a program that manages versions of the Julia kernel. Open a terminal and execute

  ```
  brew install juliaup
  ```

  After installing `juliaup`, enter the following commands in the terminal to add the `lts` (long-term stable) version of the Julia kernel.

  ```
  juliaup add lts
   juliaup default lts
  ```

  To check which version of Julia you're running, execute `juliaup status` in a terminal. It should have an asterisk in front of the `lts` version.
- Next, go to the VS Code website, and download the installer for the type of CPU architecture is on your MacBook and follow the installation

procedure.

- Open VS Code and press the `Extensions` button in the left toolbar (the four cubes symbol; `Cmd+Shift+X`). Search for and install the following three extensions: `Python`, `Julia` and `Jupyter`.
- In VS Code, go to `File` (top toolbar), and press `Open Folder` (`Cmd+K+O`). Navigate to and select the `BMLIP` folder that you downloaded from Github. This will be your development environment for the course.
- To test for a succesful software installation, open one of the lesson notebooks (in the folder `lessons/notebooks/`) and then press `Select kernel` on the top right. It should display the option `Julia`, and then `julia lts channel`. Finalize the test with `run all` (top toolbar).

Probabilistic Programming 0: Introduction to Bayesian inference

Goal

- Familiarize yourself with basic concepts from Bayesian inference such as prior and posterior distributions.
- Familiarize yourself with Jupyter notebooks and the basics of the Julia programming language.

Materials

- Mandatory
  - This notebook
  - Lecture notes on Probability Theory
  - Lecture notes on Bayesian Machine Learning
- Optional
  - Course installation guide
  - Jupyter notebook tutorial
  - Intro to programming in Julia.
  - Differences between Julia and Matlab / Python.
  - Beer Tasting Experiment
  - Savage-Dickey ratios

In 1937, one of the founders of the field of statistics, Ronald Fisher, published a story of how he explained *inference* to a friend. This story, called the "Lady Tasting Tea", has been re-told many times in different forms. In this notebook, we will re-tell one of its modern variants and introduce you to some important concepts along the way. Note that none of the material used below is new; you have all heard this in the theory lectures. The point of the Probabilistic Programming sessions is to solve practical problems so that concepts from theory become less abstract and you develop an intuition for them.

First, let's get started with activating a Julia workspace and importing some modules.

In [1]:
```
using Pkg; Pkg.activate("../../."); Pkg.instantiate();
using IJulia; try IJulia.clear_output(); catch _ end
```

Out[1]:0

Every once in a while, a cell with "code notes" is added to explain Julia-specific commands, symbols or procedures. We expect students to be proficient in at least one programming language, preferably Python, C(++) or MATLAB (most closely related to Julia), and we will not expain generic programming constructs such as control flows and data structures.

Code notes:

- The code cell above activates a specific Julia workspace (a virtual environment) that lists all packages you will need for the Probabilistic Programming sessions. The first time you run this cell, it will download and install all packages automatically.

In [2]:
```
using Distributions
using Plots
```

Code notes:

- `using` is how you import libraries and modules in Julia. Here we have imported a library of probability distributions called Distributions.jl and a library of plotting utilities called Plots.jl.

Beer Tasting Experiment

In the summer of 2017, students of the University of Amsterdam participated in a "Beer Tasting Experiment" (Doorn et al., 2019). Each participant was given two cups and were told that the cups contained Hefeweissbier, one with alcohol and one without. The participants had to taste each beer and guess which of the two contained alcohol.

We are going to do a statistical analysis of the tasting experiment. We want to know to what degree participants are able to discriminate between the alcoholic and alcohol-free beers. The Bayesian approach is about 3 core steps: (1) specifying a model, (2) absorbing the data through inference (parameter estimation), and (3) evaluating the model. We are going to walk through these steps in detail below.

1. Model Specification

Model specification consists of two parts: a likelihood function and a prior distribution.

Likelihood

A likelihood function is a function of parameters given observed data.

Here, we have an event variable X that indicates that the choice was either "correct", which we will assign the number 1, or "incorrect", which we will assign the number 0. We can model this choice with what's known as a *Bernoulli* distribution. The Bernoulli distribution is a formula to compute the probability of a binary event. It has a "rate parameter" θ, a number between 0 and 1, which governs the probability of the two events. If $\theta = 1$, then the participant will always choose the right cup ("always" = "with probability 1") and if $\theta = 0$, then the participant will never choose the right cup ("never" = "with probability 0"). Choosing at random, i.e. getting as many correct choices as incorrect choices, corresponds to $\theta = 0.5$.

As stated above, we are using the Bernoulli distribution in our tasting experiment. As the Bernoulli distribution's rate parameter θ increases, the event $X = 1$, i.e. the participant correctly guesses the alcoholic beverage, becomes more probable. The formula for the Bernoulli distribution is:

$$\begin{aligned} p(X = x \mid \theta) &= \text{Bernoulli}(x \mid \theta) \\ &= \theta^x (1 - \theta)^{1-x} \end{aligned}$$

If $X = 1$, then the formula simplifies to $p(X = 1 \mid \theta) = \theta^1 (1 - \theta)^{1-1} = \theta$. For $X = 0$, it simplifies to $p(X = 0 \mid \theta) = \theta^0 (1 - \theta)^{1-0} = 1 - \theta$. If you have multiple *independent* observations, e.g. a data set $\mathcal{D} = \{X_1, X_2, X_3\}$, you can get the probability of all observations by taking the product of individual probabilities:

$$p(\mathcal{D} \mid \theta) = \prod_{i=1}^{N} p(X_i \mid \theta)$$

As an example, suppose the first two participants have correctly guessed the beverage and a third one incorrectly guessed it. Then, the probability under $\theta = 0.8$ is

$$\begin{aligned} p(\mathcal{D} = \{1, 1, 0\} \mid \theta = 0.8) &= 0.8 \cdot 0.8 \cdot 0.2 \\ &= 0.128\,. \end{aligned}$$

That is larger than the probability under $\theta = 0.4$, which is

$$\begin{aligned} p(\mathcal{D} = \{1, 1, 0\} \mid \theta = 0.4) &= 0.4 \cdot 0.4 \cdot 0.6 \\ &= 0.096\,. \end{aligned}$$

But it is not as large as the probability under $\theta = 0.6$, which is

$$\begin{aligned} p(\mathcal{D} = \{1, 1, 0\} \mid \theta = 0.6) &= 0.6 \cdot 0.6 \cdot 0.4 \\ &= 0.144\,. \end{aligned}$$

As you can see, the likelihood function tells us how well each value of the parameter fits the observed data. In short, how "likely" each parameter value is.

Prior Distribution

In Bayesian inference, it is important to think about what kind of *prior knowledge* you have about your problem. In our tasting experiment, this corresponds to what you think the probability is that a participant will correctly choose the cup. In other words, you have some thoughts about what value θ is in this scenario. You might think that the participants' choices are all going to be roughly random. Or, given that you have tasted other types of alcohol-free beers before, you might think that the participants are going to choose the right cup most of the time. This intuition, this "prior knowledge", needs to be quantified. We do that by specifying another probability distribution for it, in this case the *Beta* distribution:

$$
\begin{aligned}
p(\theta) &= \text{Beta}(\theta \mid \alpha, \beta) \\
&= \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1}(1-\theta)^{\beta-1} .
\end{aligned}
$$

We use a Beta distribution to describe our state of knowledge about appropriate values for θ.

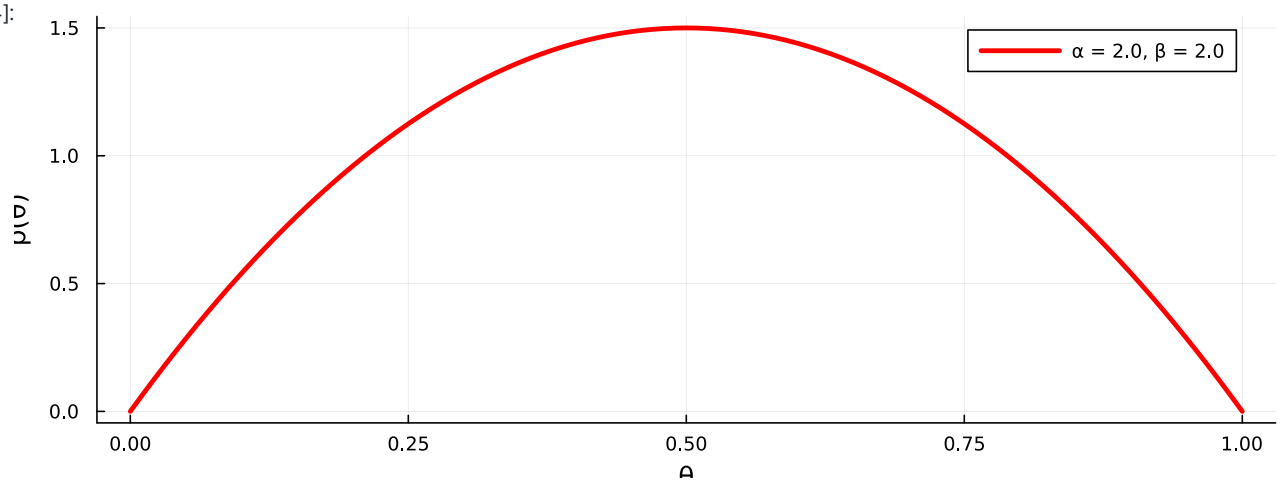
The Beta distribution computes the probability of an outcome in the interval $[0, 1]$. Like any other other distribution, it has parameters: α and β. Both are "shape parameters", meaning the distribution has a different shape for each value of the parameters. Let's visualise this!

In [3]:
```
x = [1 2 3;
     4 5 6]
```

Out[3]:
```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

In [4]:
```
# Define shape parameters
α = 2.0
β = 2.0

# Define probability distribution
pθ = Beta(α, β)

# Define range of values for θ
θ = range(0.0, step=0.01, stop=1.0)

# Visualize probability distribution function
plot(θ, pdf.(pθ, θ),
     linewidth=3,
     color="red",
     label="α = "*string(α)*", β = "*string(β),
     xlabel="θ",
     ylabel="p(θ)",
     size=(800,300))
```

Out[4]:



Code notes:

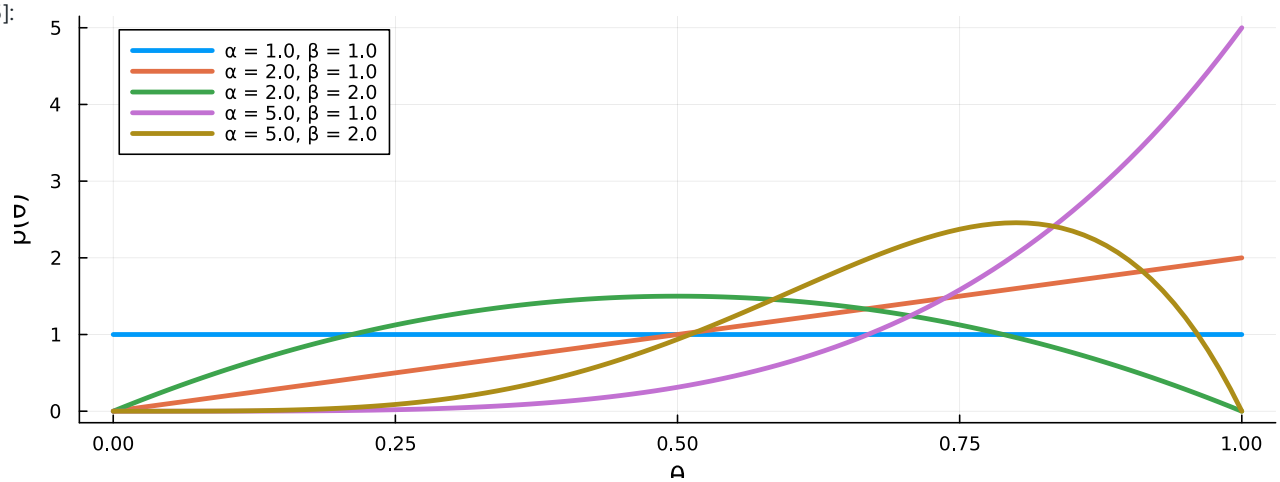
- You can use greek letters as variables (write them like in latex, e.g. \alpha, and press `tab`)
- Ranges of numbers work just like they do in Matlab (e.g. `0.0:0.1:1.0`) and Python (e.g. `range(0.0, stop=100., length=100)`). Note that Julia is strict about types, e.g. using integers vs floats.
- There is a `.` after the command `pdf`. This refers to "broadcasting": the function is applied to each element of a list or array. Here we use the `pdf` command to compute the probability for each value of θ in the array.
- Many of the keyword arguments in the `plot` command should be familiar to you if you've worked with Matplotlib (Python's plotting library).
- In the `label=` argument to plots, we have performed "string concatenation". In Julia, you write a string with double-quote characters and concatenate two strings by "multiplying", i.e. using `*`.

Note for the keen observers among you: since this is a continuous distribution, we are not actually plotting "probability", but rather "probability density" (probability densities can be larger than 1).

In [5]:
```
# Define shape parameters
α = [2.0, 5.0]
β = [1.0, 2.0]

# Define initial distribution
```

```
pθ = Beta(1.0, 1.0)

# Start initial plot
plot(θ, pdf.(pθ, θ), linewidth=3, label="α = 1.0, β = 1.0", xlabel="θ", ylabel="p(θ)", legend=:topleft)

# Loop over shape parameters
for a in α
    for b in β
        plot!(θ, pdf.(Beta(a, b), θ), linewidth=3, label="α = "*string(a)*", β = "*string(b))
    end
end
plot!(size=(800,300))
```

Out[5]:



Code notes:

- Square brackets around numbers automatically creates an Array (in Python, they create lists).
- The `:` in `:topleft` indicates a `Symbol` type. It has many uses, but here it is used synonymously with a string option (e.g. legend="topleft").
- `for` loops can be done by using a range, such as `for i = 1:10` (like in Matlab), or using a variable that iteratively takes a value in an array, such as `for i in [1,2.3]` (like in Python). More here.
- The `!` at the end of the plot command means the function is performed "in-place". In other words, it changes its input arguments. Here, we change the plot by adding lines.
- The final `plot!` is there to ensure Jupyter actually plots the figure. If you end a cell on an `end` command, Jupyter will remain silent.

As you can see, the Beta distribution is quite flexible and can capture your belief about how often participants will correctly detect the alcoholic beverage. For example, the purple line indicates that you believe that it is very probable that participants will always get it right (peak lies on $\theta = 1.0$), but you still think there is some probability that the participants will guess at random ($p(\theta = 1/2) \approx 0.3$). The yellow-brown line indicates you believe that it is nearly impossible that the participants will always get it right ($p(\theta = 1) \approx 0.0$), but you still believe that they will get it right more often than not (peak lies around $\theta \approx 0.8$).

In summary: a prior distribution $p(\theta)$ reflects our beliefs about good values for parameter θ *before* data is observed.

Exercise

Choose values for the shape parameters α and β that reflect how often you think the participants will get it right.

---

2. Parameter estimation

Now that we have specified our generative model, it is time to estimate unknown variables. We'll first look at the data and then the inference part.

Data

The data of the participants in Amsterdam is available online at the Open Science Foundation. We'll start by reading it in.

In [6]:
```
using DataFrames
using CSV
```

Code notes:

- CSV.jl is a library for reading in data stored in tables.

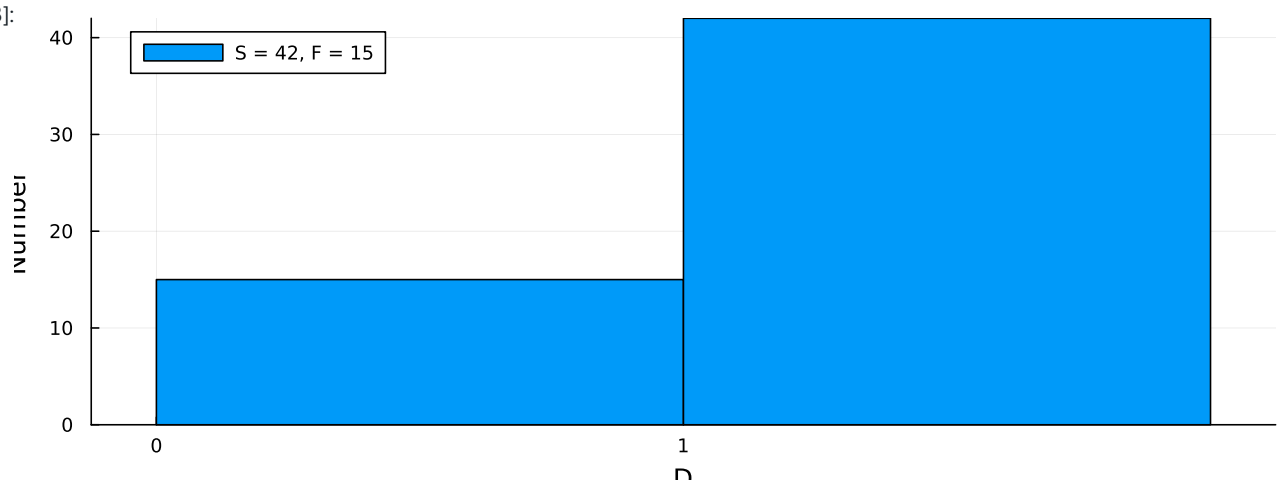
- DataFrames.jl manipulates table data (like `pandas` in Python).

In [7]:
```
# Read data from CSV file
data = DataFrame(CSV.File("../datasets/TastingBeerResults.csv"))

# Extract variable indicating correctness of guess
D = data[!, :CorrectIdentify];
println("D = ", D)
```

```
D = [1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1,
 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1]
```

In [8]:
```
# Number of successes and failures
S = sum(D .== 1)
F = sum(D .== 0)

# Visualize frequencies
histogram(D, bins=[0,1,2], label="S = "*string(S)*", F = "*string(F), xlabel="D", xticks=[0,1], ylabel="Numb
```

Out[8]:



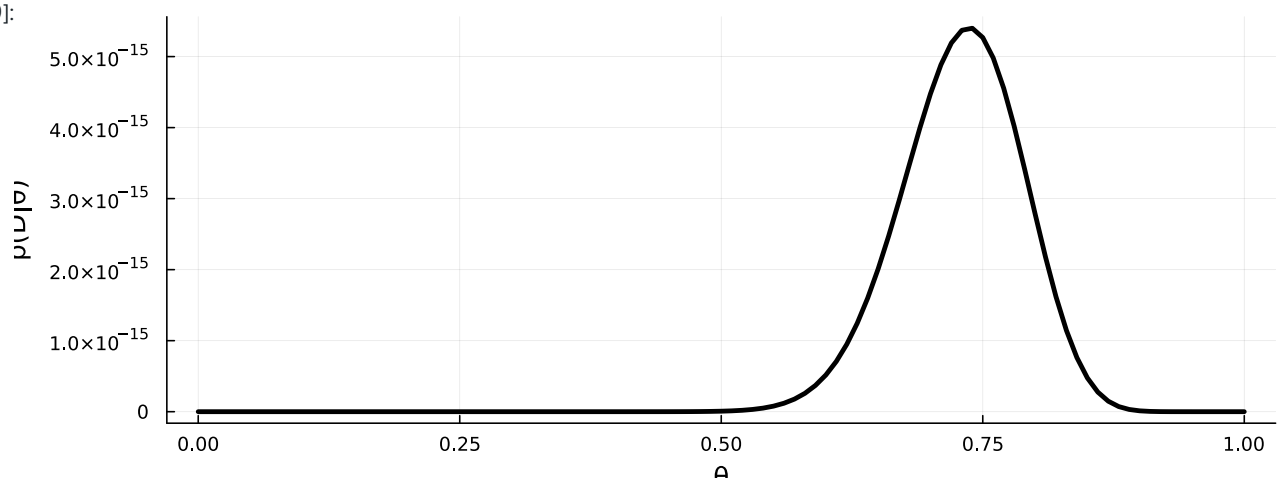
Code notes:

- The `!` in `data[!,` is specific to the DataFrames syntax.
- The `.==` checks for each element of the array D whether it is equal.

Let's visualize the likelihood of these observations.

In [9]:
```
# Define the Bernoulli likelihood function
likelihood(θ) = prod([θ^X_i * (1-θ)^(1-X_i) for X_i in D])

# Plot likelihood
plot(θ, likelihood.(θ), linewidth=3, color="black", label="", xlabel="θ", ylabel="p(D|θ)", size=(800,300))
```

Out[9]:



The likelihood has somewhat of a bell shape, peaking just below $\theta = 0.75$. Note that the y-axis is very small. Indeed, the likelihood is not a proper probability distribution, because it doesn't integrate / sum to 1.

Inference

Using our generative model, we can estimate parameters for unknown variables. Remember Bayes' rule:

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta) p(\theta)}{p(\mathcal{D})}\,.$$

The posterior $p(\theta \mid \mathcal{D})$ equals the likelihood $p(\mathcal{D} \mid \theta)$ times the prior $p(\theta)$ divided by the evidence $p(\mathcal{D})$. In our tasting experiment, we have a special thing going on: conjugacy. The Beta distribution is "conjugate" to the Bernoulli likelihood, meaning that the posterior distribution is also going to be a Beta distribution. Specifically with the Beta-Bernoulli combination, it is easy to see what conjugacy actually means. Recall the formula for the Beta distribution:

$$p(\theta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha - 1} (1 - \theta)^{\beta - 1}\,.$$

The term $\Gamma(\alpha + \beta) / \left(\Gamma(\alpha)\Gamma(\beta)\right)$ normalises this distribution. If you ignore that and multiply it with the likelihood, you get something that simplifies beautifully:

$$\begin{aligned} p(\mathcal{D} \mid \theta) p(\theta) \;&\propto\; \prod_{i=1}^{N} \left[\theta^{X_i} (1 - \theta)^{1 - X_i}\right] \cdot \theta^{\alpha - 1} (1 - \theta)^{\beta - 1} \\ &=\; \theta^{\sum_{i=1}^{N} X_i} (1 - \theta)^{\sum_{i=1}^{N} 1 - X_i} \cdot \theta^{\alpha - 1} (1 - \theta)^{\beta - 1} \\ &=\; \theta^{S} (1 - \theta)^{F} \cdot \theta^{\alpha - 1} (1 - \theta)^{\beta - 1} \\ &=\; \theta^{S + \alpha - 1} (1 - \theta)^{F + \beta - 1}\,, \end{aligned}$$

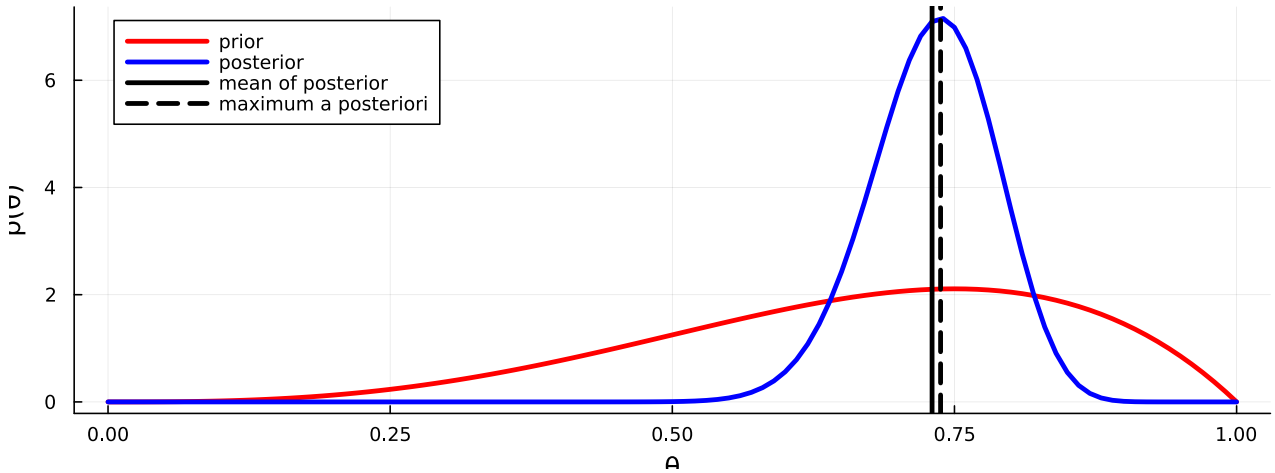
where $S = \sum_{i=1}^{N} X_i$ is the number of successes (correct guesses) and $F = \sum_{i=1}^{N} 1 - X_i$ is the number of failures (incorrect guesses).

This last line is again the formula for the Beta distribution (except for a proper normalisation) but with different parameters ($S + \alpha$ instead of α and $F + \beta$ instead of β). This is what we mean by conjugacy: applying Bayes rule to a conjugate prior and likelihood pair yields a posterior distribution of the same family as the prior, which in this case is a Beta distribution.

Let's now visualise the posterior after observing the data from Amsterdam.

In [10]:
```
# Define shape parameters of prior distribution
α0 = 4.0
β0 = 2.0

# Define prior distribution
pθ = Beta(α0, β0)

# Update parameters for the posterior
αN = α0 + sum(D .== 1)
βN = β0 + sum(D .== 0)

# Define posterior distribution
pθD = Beta(αN, βN)

# Mean of posterior
mean_post = αN / (αN + βN)
mode_post = (αN - 1) / (αN + βN - 2)

# Visualize probability distribution function
plot(θ, pdf.(pθ, θ), linewidth=3, color="red", label="prior", xlabel="θ", ylabel="p(θ)")
plot!(θ, pdf.(pθD, θ), linewidth=3, color="blue", label="posterior", size=(800,300))
vline!([mean_post], color="black", linewidth=3, label="mean of posterior", legend=:topleft)
vline!([mode_post], color="black", linewidth=3, linestyle=:dash, label="maximum a posteriori", legend=:topl
```

Out[10]:



Code notes:

- `vline` draw a vertical line in the plot, at the specified point on the x-axis.
- `mode()` extracts the mode) of the supplied distribution, i.e. the point with the largest probability.

That looks great! We have updated our belief from a very broad prior to a much sharper posterior.

The posterior contains a lot of information: it tells us something about every value for θ. Sometimes, we are interested in a point estimate, i.e. the probability of a single value for θ under the posterior. Two well-known point estimators are the mean of the posterior and the mode (the value for θ with the highest probability). I have plotted both point estimates in the figure above. In this case, they are nearly equal.

Exercise

Plug the shape parameters of your prior into a copy of the cell above and see how your posterior differs.

---

3. Model Evaluation

Given our model assumptions and a posterior for theta, we can now make quantitative predictions about how well we think people can recognize alcoholic from non-alcoholic hefeweizen. But suppose you meet someone else who is absolutely sure that people can't tell the difference. Can you say something about the probability of his belief, given the experiment?

Technically, this is a question about comparing the performance of different models. Model comparison is also known in the statistical literature as "hypothesis testing".

In hypothesis testing, you start with a null hypothesis $\mathcal{H}_0$, which is a particular choice for the detection parameter θ. In the question above, the other person's belief corresponds to $\theta = 0.5$. We then have an alternative hypothesis $\mathcal{H}_1$, namely that his belief is wrong, i.e. $\theta \neq 0.5$. From a Bayesian perspective, hypothesis testing is just about comparing the posterior beliefs about these two hypotheses:

$$\underbrace{\frac{p(\mathcal{H}_1|\mathcal{D})}{p(\mathcal{H}_0|\mathcal{D})}}_{\text{Posterior belief over hypotheses}} = \underbrace{\frac{p(\mathcal{H}_1)}{p(\mathcal{H}_0)}}_{\text{Prior belief over hypotheses}} \times \underbrace{\frac{p(\mathcal{D}|\mathcal{H}_1)}{p(\mathcal{D}|\mathcal{H}_0)}}_{\text{Likelihood of hypotheses}} .$$

Note that the evidence term $p(\mathcal{D})$ is missing, because it appears in the posterior for both hypotheses and therefore cancels out. The hypothesis likelihood ratio is also called the **Bayes factor**. Bayes factors can be hard to compute, but in some cases we can simplify it: if the null hypothesis is a specific value of interest, for instance $\theta = 0.5$, and the alternative hypothesis is not that specific value, e.g. $\theta \neq 0.5$, then the factor reduces to what's known as a Savage-Dickey Ratio (see Appendix A of Wagemakers et al., 2010):

$$\frac{p(\mathcal{D}|\mathcal{H}_1)}{p(\mathcal{D}|\mathcal{H}_0)} = \frac{p(\theta = 0.5)}{p(\theta = 0.5 \mid \mathcal{D})} .$$

This compares the probability of $\theta = 0.5$ under the prior versus $\theta = 0.5$ under the posterior. It effectively tells you how much your belief changes after observing the data. Let's compute the Savage-Dickey ratio for our experiment:

In [11]:

```
BF_10 = pdf(p0, 0.5) / pdf(p0D, 0.5)
println("The Bayes factor for H1 versus H0 = "*string(BF_10))
```

```
The Bayes factor for H1 versus H0 = 229.23178145896927
```

So, in the experiment, the alternative hypothesis "*students can discriminate alcoholic from non-alcoholic Hefeweissbier*" is more than 200 times more probable than the null hypothesis that "*students cannot discriminate alcoholic from non-alcoholic Hefeweissbier*".

Exercise

Compute the Bayes factor for your prior and posterior distribution. How many times is the alternative hypothesis more probable than the null hypothesis?

In [ ]:

Probabilistic Programming 1: Bayesian inference in conjugate models

Goal

- Practice forming factor graphs out of probabilistic models.
- Familiarize yourself with message passing on factor graphs.
- Practice specifying models and inference procedures in a probabilistic programming language.

Materials

- Mandatory
  - This notebook
  - Lecture notes on factor graphs
  - Lecture notes on continuous data
  - Lecture notes on discrete data
- Optional
  - Chapters 2 and 3 of Model-Based Machine Learning.
  - Differences between Julia and Matlab / Python.

In [1]:

```
using Pkg; Pkg.activate("../../."); Pkg.instantiate();
using IJulia; try IJulia.clear_output(); catch _ end
```

Out[1]:0

In [2]:

```
using CSV
using Random
using DataFrames
using LinearAlgebra
using SpecialFunctions
using Distributions
using ReactiveMP
using RxInfer
using Plots
default(label="", linewidth=4, margin=10Plots.pt)

import CairoMakie: tricontourf
import ReactiveMP: @call_rule, prod
import BayesBase: ClosedProd
```

Problem: A Job Interview

Suppose you have graduated and applied for a job at a tech company. The company wants a talented and skilled employee, but measuring a person's skill is tricky; even a highly-skilled person makes mistakes and - vice versa - people with few skills can get lucky. They decide to approach this objectively and construct a statistical model of responses.

In this session, we will look at estimating parameters in various distributions under the guise of assessing skills based on different types of interview questions. We will practice message passing on factor graphs using a probabilistic programming language developed at the TU/e: RxInfer.jl.

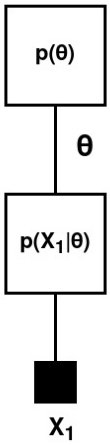
1: Right or wrong

To start, the company wants to test the applicants' programming skills and created a set of bug detection questions. We will first look at a single question, which we treat as an outcome variable X_1. Your answer is either right or wrong, which can be modelled with a Bernoulli likelihood function. The company assumes you have a skill level, denoted θ, and the higher the skill, the more likely you are to get the question right. Since

the company doesn't know anything about you, they chose an uninformative prior distribution: the Beta(1,1). We can write the generative model for answering this question as follows:

$$\begin{aligned} p(X_1, \theta) &= p(X_1 \mid \theta) \cdot p(\theta) \\ &= \text{Bernoulli}(X_1 \mid \theta) \cdot \text{Beta}(\theta \mid \alpha = 1, \beta = 1)\,. \end{aligned}$$

The factor graph for this model is:



We are now going to construct this factor graph / probabilistic model in RxInfer.

In [3]:
```
@model function beta_bernoulli1(X)
    "Beta-Bernoulli model with single observation"

    # Prior distribution
    θ ~ Beta(1.0, 1.0)

    # Likelihood of data point
    X ~ Bernoulli(θ)
end
```

Note that we may define random variables using a tilde symbol, which should be read as "[random variable] is distributed according to [probability distribution function]". For example, $\theta \sim \text{Beta}(1, 1)$ should be read as "θ is distributed according to a Beta(θ | a=1, b=1) probability distribution".

Having defined the model, we can now call an inference procedure which will automatically compute the posterior distribution for the random variable:

In [4]:
```
results = infer(
    model      = beta_bernoulli1(),
    data       = (X = 1,),
)
```

Out[4]:
```
Inference results:
  Posteriors       | available for (θ)
```

Under the hood, RxInfer is performing message passing. Each variable definition actually creates a factor node and each node will send a message. The collision of messages will automatically update the marginal distributions.

We may inspect some of the message and marginal computations with the following commands:

In [5]:
```
message1 = @call_rule Beta(:out, Marginalisation) (m_a = PointMass(1.0), m_b = PointMass(1.0))
```

Out[5]:
```
Beta{Float64}(α=1.0, β=1.0)
```

In [6]:
```
message2 = @call_rule Bernoulli(:p, Marginalisation) (m_out = PointMass(1),)
```

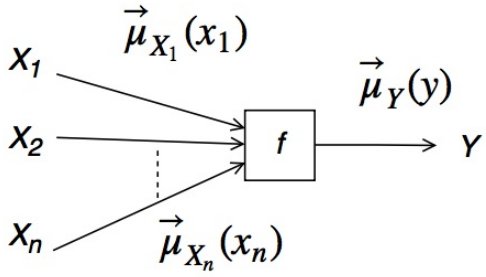
Out[6]:
```
Beta{Float64}(α=2.0, β=1.0)
```

Alright. So, they are both Beta distributions. Do they actually make sense? Where do these parameters come from?

Recall from the lecture notes that the formula for messages sent by factor nodes is:

$$\begin{aligned} \underbrace{\overrightarrow{\mu}_Y(y)}_{\text{outgoing message}} &= \sum_{x_1,\ldots,x_n} \underbrace{\overrightarrow{\mu}_{X_1}(x_1) \cdots \overrightarrow{\mu}_{X_n}(x_n)}_{\text{incoming messages}} \\ &\quad \cdot \underbrace{f(y, x_1, \ldots, x_n)}_{\text{node function}}\,, \end{aligned}$$

visually represented by



The prior node is not connected to any other unknown variables and so does not receive incoming messages. Its outgoing message is

$$\begin{aligned}\mu_1(\theta) &= f(\theta) \\ &= \text{Beta}(\theta \mid \alpha = 1, \beta = 1)\,.\end{aligned}$$

We can also derive the message from the likelihood node by hand. For this, we need to know that the message coming from the observation $\overleftarrow{\mu}(x)$ is a delta function, which, if you gave the right answer ($X_1 = 1$), has the form $\delta(X_1 - 1)$. The "node function" is the Bernoulli likelihood $\text{Bernoulli}(X_1 \mid \theta)$. Another thing to note is that this is essentially a convolution with respect to a delta function and that its sifting property holds:

$$\int_X \delta(X - x)\; f(X, \theta) \mathrm{d}X = f(x, \theta)\,.$$

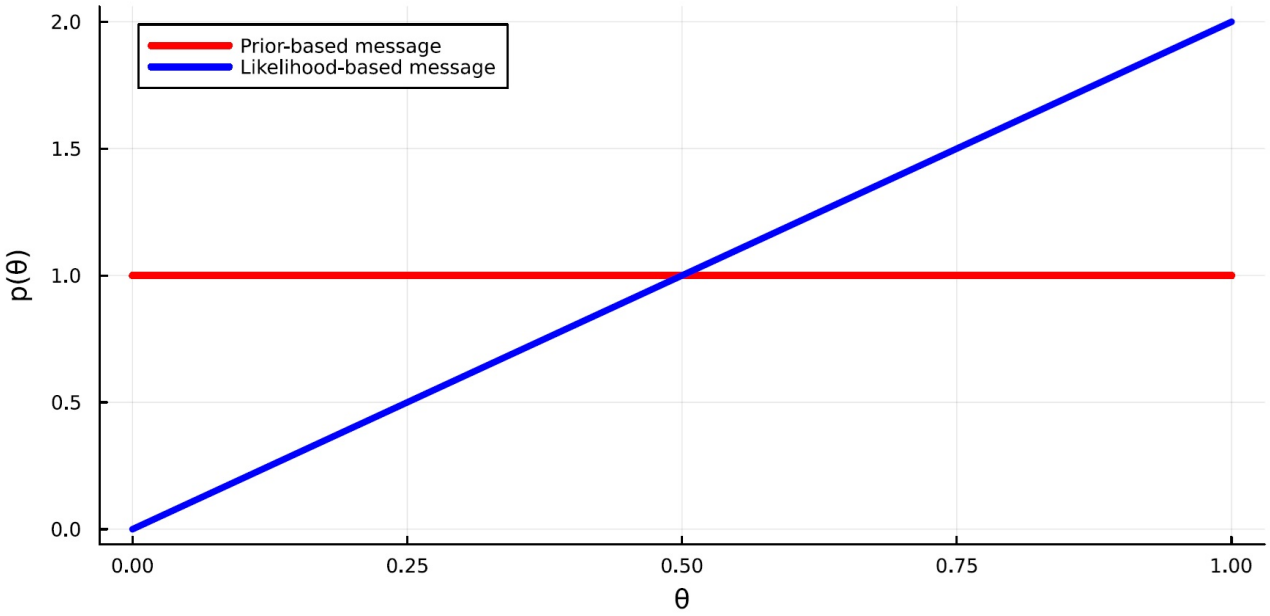
The fact that X_1 is a discrete variable instead of a continuous one, does not negate this. Using these facts, we can perform the message computation by hand:

$$\begin{aligned}\mu_2(\theta) &= \sum_{X_1} \mu(X_1)\; f(X_1, \theta) \\ &= \sum_{X_1} \delta(X_1 - 1)\; \text{Bernoulli}(X_1 \mid \theta) \\ &= \sum_{X_1} \delta(X_1 - 1)\; \theta^{X_1}(1-\theta)^{1-X_1} \\ &= \theta^1(1-\theta)^{1-1}\,.\end{aligned}$$

Remember that the pdf of a Beta distribution is proportional to $\theta^{\alpha-1}(1-\theta)^{\beta-1}$. So, if you read the second-to-last line above as $\theta^{2-1}(1-\theta)^{1-1}$, then the outgoing message $\overleftarrow{\mu}(\theta)$ is proportional to a Beta distribution with $\alpha = 2$ and $\beta = 1$. So, our manual derivation matches RxInfer's message 2.

Let's now look at these messages visually.

In [7]:
```
# Sample space of random variable
θ_range = range(0, step=0.01, stop=1.0)

# Plot messages
plot( θ_range, x -> pdf.(message1, x), color="red", label="Prior-based message", xlabel="θ", ylabel="p(θ)")
plot!(θ_range, x -> pdf.(message2, x), color="blue", label="Likelihood-based message", legend=:topleft, size
```



The marginal distribution for θ, representing the posterior $p(\theta \mid X_1)$, is obtained by taking the product (followed by normalization) of the two messages: $\mu_1(\theta) \cdot \mu_2(\theta)$. Multiplying two Beta distributions produces another Beta distribution with parameter:

$$\begin{aligned} \alpha &\leftarrow \alpha_1 + \alpha_2 - 1 \\ \beta &\leftarrow \beta_1 + \beta_2 - 1\,, \end{aligned}$$

In our case, the new parameters would be $\alpha = 1 + 2 - 1 = 2$ and $\beta = 1 + 1 - 1 = 1$.

Let's check with RxInfer. The product of the two Beta's can be computed with:

In [8]:
```
prod(ClosedProd(), Beta(1.,1.), Beta(1.,1.))
```

Out[8]:
```
Beta{Float64}(α=1.0, β=1.0)
```

Extra information:

The `ClosedProd()` input indicates that julia should not use the generic `prod` function (e.g., for products of `Float64`'s or `Int64`'s), but that it should use the product operations defined by the RxInfer ecosystem for parametric probability distributions. It is an example of Julia's "multiple dispatch" feature, which is making waves in the programming languages world (youtube, blog).

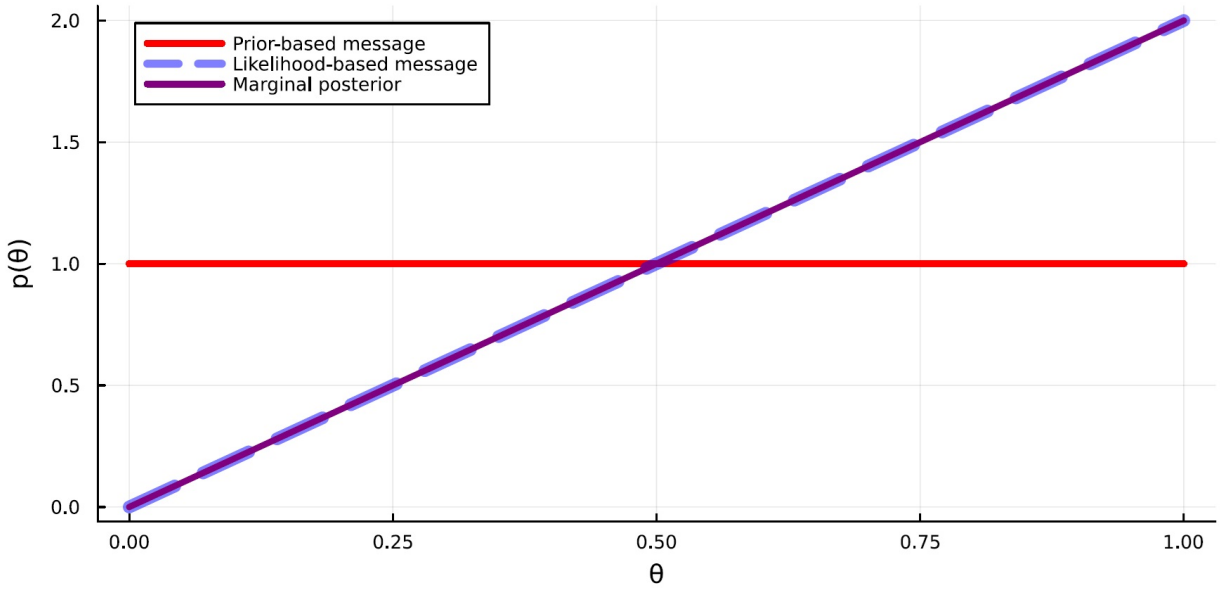
That matches our manual derivations as well as the posterior reported by the `inference` procedure:

In [9]:
```
posterior = results.posteriors[:θ]
```

Out[9]:
```
Beta{Float64}(α=2.0, β=1.0)
```

Let's visualize the messages as well as the marginal posterior.

In [10]:
```
plot( θ_range, x -> pdf.(message1, x), color="red", label="Prior-based message", xlabel="θ", ylabel="p(θ)")
plot!(θ_range, x -> pdf.(message2, x), color="blue", linewidth=8, linestyle=:dash, alpha=0.5, label="Likeli
plot!(θ_range, x -> pdf.(posterior, x), color="purple", label="Marginal posterior")
```



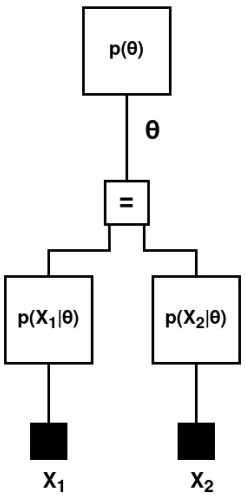
The pdf of the marginal distribution lies on top of the pdf of Message 2. That's not always going to be the case; the Beta(1,1) distribution is special in that when you multiply Beta(1,1) with a general Beta(a,b) the result will always be Beta(a,b), kinda like multiplying by 1. We call prior distributions that have this special effect "non-informative priors".

Multiple questions

Of course, in practice you would be evaluated on multiple questions, which are essentially more samples from the underlying distribution that is your skill level. We are going to add question outcomes to the model. For now, we will still work with right-or-wrong questions (i.e., binary outcomes), denoted $X = (X_1, \ldots, X_N)$. The generative model becomes

$$\begin{aligned} p(X, \theta) &= p(\theta) \prod_{i=1}^{N} p(X_i \mid \theta) \\ &= \text{Beta}(\theta) \prod_{i=1}^{N} \text{Bernoulli}(X_i \mid \theta)\,, \end{aligned}$$

The factor graph for this model is:



Specified in code, this is:

In [11]:
```
@model function beta_bernoulli(X,N)
    "Beta-Bernoulli model with multiple observations"

    # Prior distribution
    θ ~ Beta(3.0, 2.0)
```

```
        # Loop over data
        for i in 1:N

            # Likelihood of i-th data points
            X[i] ~ Bernoulli(θ)

        end
    end
```

You may have noticed that the prior distribution changed; the company now assumes that you must have *some* skill if you applied for the position. This is reflected in the prior Beta distribution with $\alpha = 3.0$ and $\beta = 2.0$.

Now suppose we have two outcomes, $X_1 = 1$ and $X_2 = 0$:

In [12]:
```
X = [1; 0];
N = length(X)
```

Out[12]:2

Running the inference procedure is nearly exactly the same, except now we have to provide the sample size parameter N:

In [13]:
```
results = infer(
    model = beta_bernoulli(N=N),
    data  = (X = X,),
)
```

Out[13]:
```
Inference results:
  Posteriors       | available for (θ)
```

We now have two likelihood-based messages:

In [14]:
```
message1 = @call_rule Bernoulli(:p, Marginalisation) (m_out = PointMass(X[1]),)
```

Out[14]:Beta{Float64}(α=2.0, β=1.0)

In [15]:
```
message2 = @call_rule Bernoulli(:p, Marginalisation) (m_out = PointMass(X[2]),)
```

Out[15]:Beta{Float64}(α=1.0, β=2.0)

Taking their product gives us a total likelihood message, i.e.,

$$\begin{aligned}\mu_3(\theta) &= \mu_1(\theta) \cdot \mu_2(\theta) \\ &= \sum_{X_1} \delta(X_1 - 1)\ \text{Bernoulli}(X_1 \mid \theta) \cdot \\ &\quad \sum_{X_2} \delta(X_2 - 0)\ \text{Bernoulli}(X_2 \mid \theta) \\ &= \text{Beta}(\alpha = 2, \beta = 1) \cdot \text{Beta}(\alpha = 1, \beta = 2) \\ &= \text{Beta}(\alpha = 2, \beta = 2)\end{aligned}$$

Let's verify that manual calculation using RxInfer:

In [16]:
```
message3 = prod(ClosedProd(), message1, message2)
```

Out[16]:Beta{Float64}(α=2.0, β=2.0)

This product of messages is the result of passing the two likelihood-based messages through an equality node (see Bert's lecture):

$$\begin{aligned}\mu_3(\theta) &= \int_{\theta'} \int_{\theta''} \overrightarrow{\mu}(\theta'')\ f_=(\theta, \theta', \theta'')\ \overleftarrow{\mu}(\theta') \mathrm{d}\theta'\ \mathrm{d}\theta'' \\ &= \mu'(\theta) \cdot \mu''(\theta)\,.\end{aligned}$$

You don't have to worry about explicitly managing equality nodes; most packages automatically perform these operations (or functionally similar ones) under the hood.

Exercise

What would be your likelihood-based message if your data was $X = [0\ \ 0\ \ 0]$?

In [ ]:

Now that we have a likelihood-based message, we can combine that with the message from the prior distribution, $\mu_4(\theta) = \text{Beta}(\alpha = 3, \beta = 2)$, to get the marginal posterior for θ:

$$
\begin{aligned}
p(\theta \mid X_1, X_2) &= \mu_3(\theta) \cdot \mu_4(\theta) \\
&= \text{Beta}(\alpha = 2, \beta = 2) \cdot \text{Beta}(\alpha = 3, \beta = 2) \\
&= \text{Beta}(\alpha = 4, \beta = 3)\,.
\end{aligned}
$$

Let's check with RxInfer:

In [17]:
```
message4 = Beta(3.0, 2.0)
posterior = prod(ClosedProd(), message3, message4)
```

Out[17]:`Beta{Float64}(α=4.0, β=3.0)`
That should also be equal to the inferred posterior:

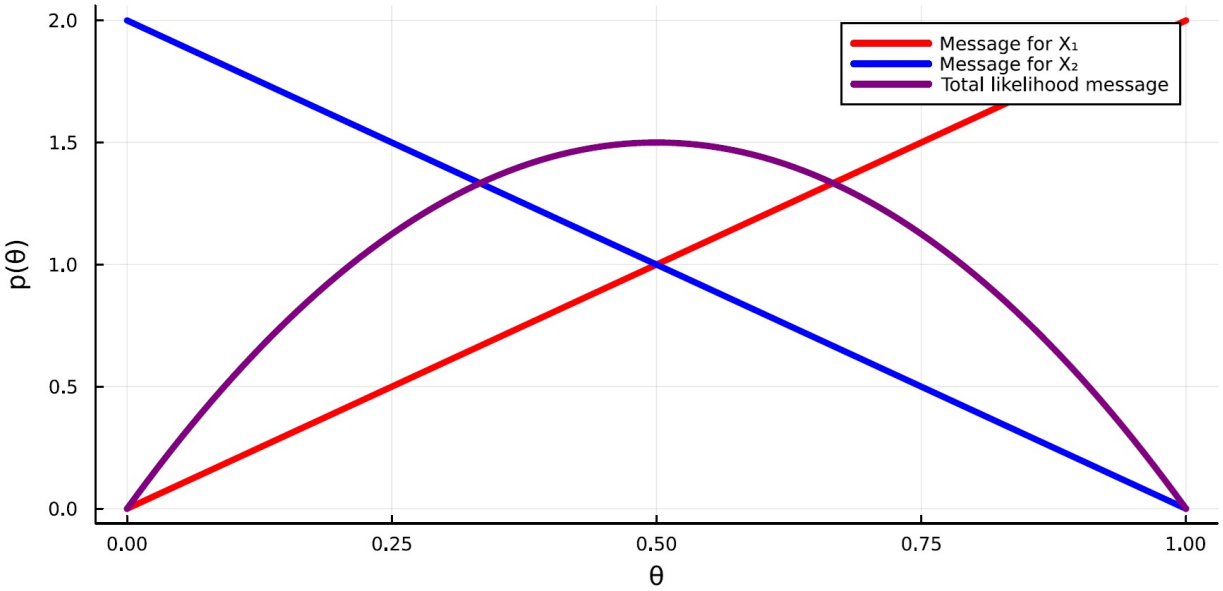
In [18]:
```
results.posteriors[:θ]
```

Out[18]:`Beta{Float64}(α=4.0, β=3.0)`
Great. That checks out.

Let's also visualize the messages and the resulting marginal:

In [19]:
```
plot( θ_range, x -> pdf.(message1, x), color="red", label="Message for X₁", xlabel="θ", ylabel="p(θ)")
plot!(θ_range, x -> pdf.(message2, x), color="blue", label="Message for X₂", size=(800,400))
plot!(θ_range, x -> pdf.(message3, x), color="purple", label="Total likelihood message")
```

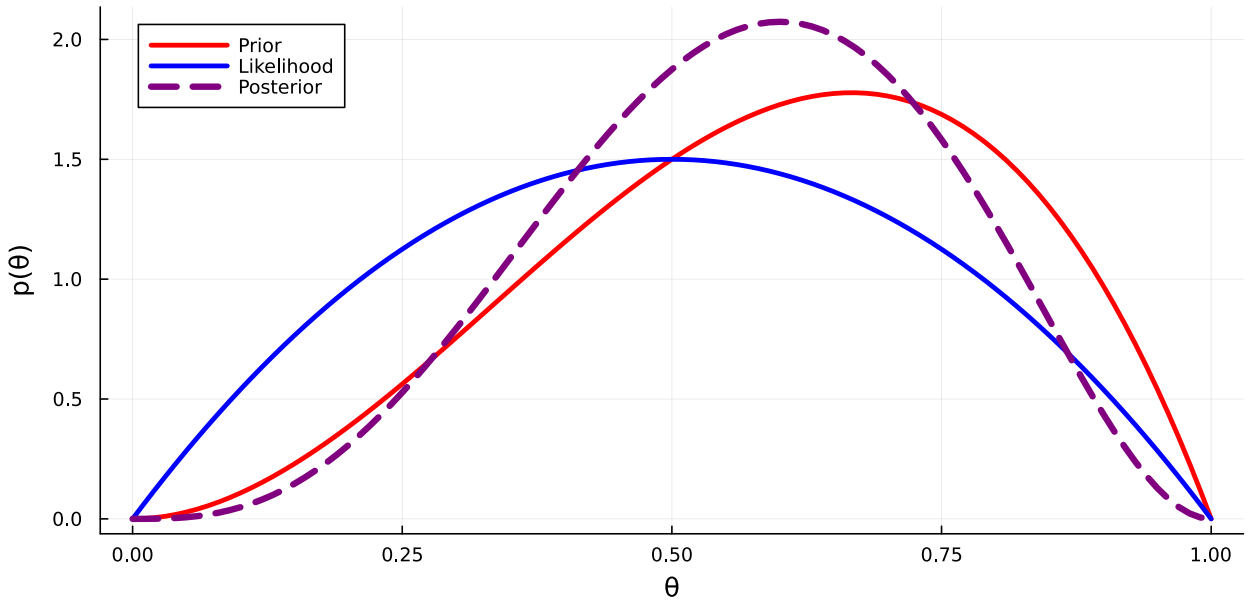
Out[19]:



Message 1 and message 2 are direct opposites: the first increases the estimate and the second decreases the estimate of your skill level. The total likelihood message ends up being centered on the average, i.e., 0.5. If we plot the prior- and likelihood-based messages as well as the marginal, we can see that Bayes' rule is really a weighted average.

In [20]:
```
# Plot prior-based message
plot( θ_range, x -> pdf(message4, x), color="red", linewidth=3, label="Prior", xlabel="θ", ylabel="p(θ)")

# plot likelihood-based message
plot!(θ_range, x -> pdf(message3, x), color="blue", linewidth=3, label="Likelihood", size=(800,400))

# Plot marginal posterior
plot!(θ_range, x -> pdf(posterior, x), color="purple", linewidth=4, linestyle=:dash, label="Posterior")
```

Out[20]:



2. Score questions

Suppose you are not tested on a right-or-wrong question, but on a score question. For instance, you have to complete a piece of code for which you get a score. If all of it was wrong you get a score of 0, if some of it was correct you get a score of 1 and if all of it was correct you get a score 2. That means we have a likelihood with three outcomes: $X_1 = \{0, 1, 2\}$. Suppose we once again ask two questions, X_1 and X_2. The order in which we ask these questions does not matter, so that means we choose Categorical distributions for these likelihood functions: $X_1, X_2 \sim \text{Categorical}(\theta)$. The parameter θ is no longer a single parameter, indicating the probability of getting the question right, but a vector of three parameters: $\theta = (\theta_1, \theta_2, \theta_3)$. Each θ_k indicates the probability of getting the k-th outcome. In other words, θ_1 indicates the probability of getting 0 points, θ_2 of getting 1 point and θ_3 of getting 2 points. A highly-skilled applicant mights have a parameter vector of $(0.05, 0.1, 0.85)$, for example. The prior distribution conjugate to the Categorical distribution is the Dirichlet distribution.

Let's look at the generative model:

$$p(X_1, X_2, \theta) = p(X_1 \mid \theta)p(X_2 \mid \theta)p(\theta)\,.$$

It's the same as before. The only difference is the parameterization of the distributions:

$$\begin{aligned} p(X_1 \mid \theta) &= \text{Categorical}(X_1 \mid \theta) \\ p(X_2 \mid \theta) &= \text{Categorical}(X_2 \mid \theta) \\ p(\theta) &= \text{Dirichlet}(\theta \mid \alpha)\,, \end{aligned}$$

where α are the concentration parameters of the Dirichlet. This model can be written directly in RxInfer:

In [21]:
```
@model function dirichlet_categorical(X, N, α)

    # Prior distribution
    θ ~ Dirichlet(α)

    # Likelihood
    for i in 1:N

        X[i] ~ Categorical(θ)

    end
end
```

Suppose you got a score of 1 on the first question, a score of 2 on the second question and a score of 2 on the third question. In a one-hot encoding, this is represented as:

In [22]:
```
X = [[0, 1, 0],
     [0, 0, 1],
     [0, 0, 1]];

N = length(X);
```

Exercise

Compute the likelihood-based message towards θ. I've given the three messages below:

In [23]:
```
message1 = @call_rule Categorical(:p, Marginalisation) (q_out = PointMass(X[1]),);
message2 = @call_rule Categorical(:p, Marginalisation) (q_out = PointMass(X[2]),);
message3 = @call_rule Categorical(:p, Marginalisation) (q_out = PointMass(X[3]),);
```

In [ ]:

The company thinks that applicants are more likely to get the answer partially correct than entirely wrong or entirely right. This is reflected in their prior concentration parameters:

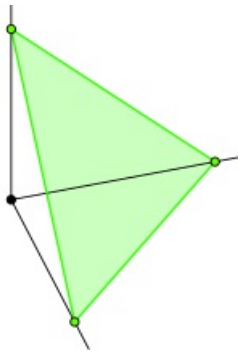
In [24]:
```
# Prior concentration parameters
α0 = [1.0, 2.0, 1.0];
```

The inferrred posterior is a Dirichlet distribution with higher concentrations for scores 1 and 2:

In [25]:
```
results = infer(
    model = dirichlet_categorical(α=α0, N=N),
    data = (X = X,),
)
```

Out[25]:
```
Inference results:
  Posteriors       | available for (θ)
```

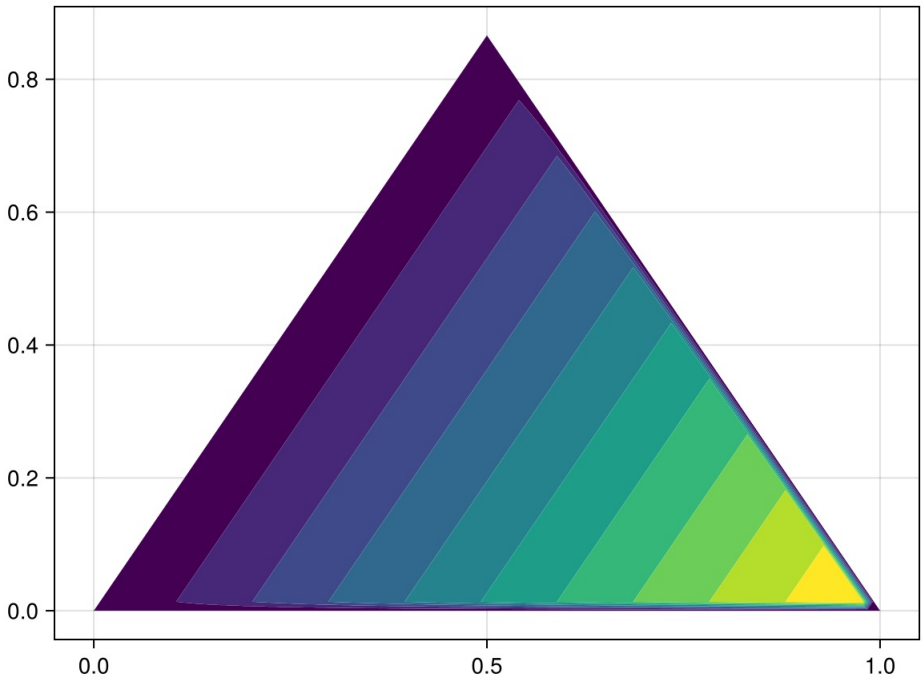
Visualizing a Dirichlet distribution is a bit tricky. In the special case of 3 parameters, we can plot the probabilities on a simplex. As a reminder, a simplex in 3-dimensions is the triangle between the coordinates $[0, 0, 1]$, $[0, 1, 0]$ and $[1, 0, 0]$:



Every point on that triangle is 3D vector that sums to 1. Since the triangle is a 2-dimensional subspace, we can map the 3D simplex to a 2D triangular surface and plot the Dirichlet's probability density over it.

In [26]:
```
# Load pre-generated triangular mesh
mesh = Matrix(DataFrame(CSV.File("../datasets/trimesh.csv")))

# Compute probabilities on trimesh of simplex
pvals = [pdf(Dirichlet(α0), mesh[n,3:5]) for n in 1:size(mesh,1)]

# Generate filled contour plot
tricontourf(mesh[:,1], mesh[:,2], pvals)
```

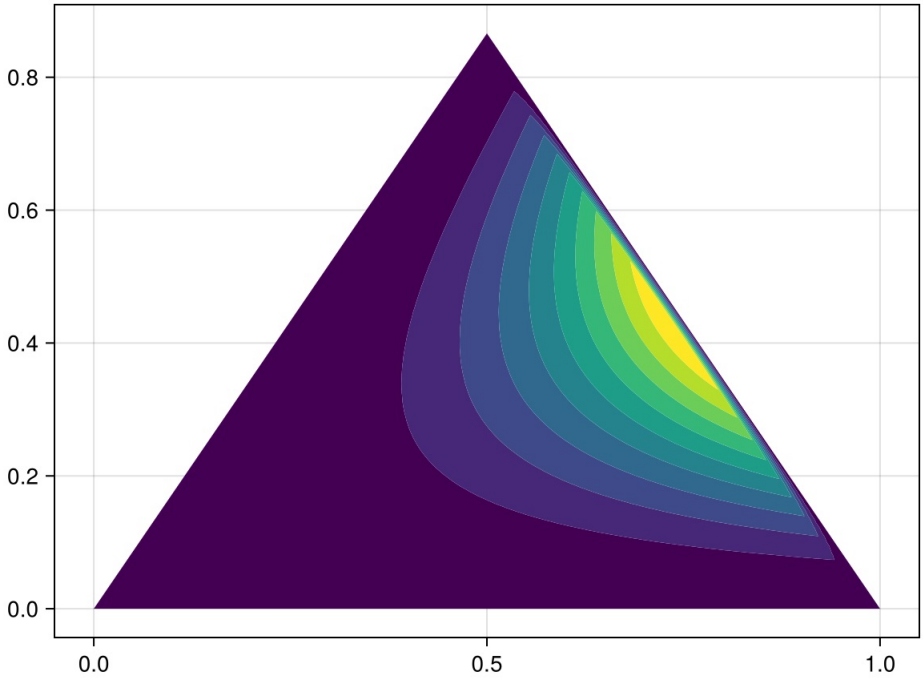
Out[26]:



The yellow spot is the area of high probability, with the contour lines indicating regions of decreasing probability. These prior concentration parameters clearly indicate a higher density in one corner of the simplex. Let's inspect the posterior concentration parameters.

In [27]:
```
# Extract parameters
αN = params(results.posteriors[:θ])[1]

# Compute probabilities on trimesh of simplex
pvals = [pdf(Dirichlet(αN), mesh[n,3:5]) for n in 1:size(mesh,1)]

# Generate filled contour plot
tricontourf(mesh[:,1], mesh[:,2], pvals)
```

Out[27]:



The distribution has shifted to the upper right edge and has concentrated more (i.e., the probability contours drop off more rapidly).

Exercise

Add some more data to the model. What scores lead to a yellow blob in the exact middle of the simplex?

In [ ]:

---

3. Continuous-valued score

Suppose the company wants to know how fast applicants respond to questions. The interview conductor also has a stopwatch and measures your response time per question. Each applicant is assumed to have some underlying response speed θ. Each measurement X_i is a noisy observation of that response speed, where the noise is assumed to be symmetric, i.e., the applicant might a bit as faster as often as they are a bit slower than usual. The Gaussian, or Normal, distribution is a symmetric continuous-valued distribution and will characterize the assumption well. The likelihood is therefore:

$$p(X \mid \theta) = \mathcal{N}(X \mid \theta, \sigma^2)\,,$$

where σ is the standard deviation. The conjugate prior to the mean in a Gaussian likelihood is another Gaussian distribution:

$$p(\theta) = \mathcal{N}(\theta \mid m_0, v_0)$$

with m_0, v_0 as prior mean and variance.

Exercise

Write down the full generative model using the above prior distribution and the likelihood of N observations.

In code, the model is:

In [28]:
```
@model function normal_normal(X, m0, v0, σ, N)

    # Prior distribution
    θ ~ Normal(mean = m0, variance = v0)

    # Likelihood
    for i = 1:N

        X[i] ~ Normal(mean = θ, variance = σ^2)

    end
end
```

The interview conductor cannot stop immediately after you have responded. From previous interviews, the company knows that the conductor in front of you is typically off by roughly 2 seconds. That translates to a likelihood variance of $\sigma^2 = 4$.

In [29]:
```
σ = 2.0;
```

Your response times on the questions are:

In [30]:
```
X = [ 52.390036995147426
      74.49846899398719
      50.92640384934159
      39.548361884989717];

N = length(X);
```

The company designed the questions such that they think it may take the average participant 60 seconds to respond, $\pm$ 20 seconds. That translates to the following values for the prior parameters:

In [31]:
```
m0 = 60;
v0 = 20;
```

In [32]:
```
results = infer(
    model = normal_normal(m0=m0, v0=v0, σ=σ, N=N),
    data  = (X = X,),
)

posterior = results.posteriors[:θ]
mean_var(posterior)
```

Out[32]: `(54.61030279130141, 0.9523809523809523)`

Ah! It seems that you are a bit faster than the average participant.

Let's visualize the prior message, the total likelihood message and the posterior again. First, we want to get the likelihood message:

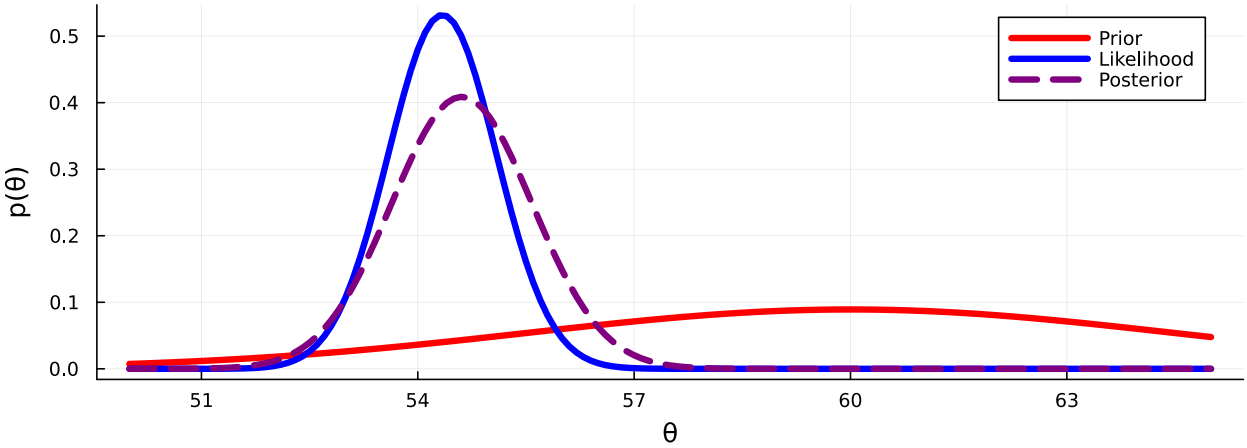
In [33]:
```
message = @call_rule NormalMeanVariance(:μ, Marginalisation) (m_out=PointMass(X[1]), m_v=PointMass(1.5^2))
for i in 2:N
```

```
    message_i = @call_rule NormalMeanVariance(:μ, Marginalisation) (m_out=PointMass(X[i]), m_v=PointMass(1.
    message = prod(ClosedProd(), message, message_i)
end
mean_var(message)
```

Out[33]: (54.34081793086648, 0.5625)

In [34]:
```
# Range of values to plot pdf for
θ_range = range(50.0, step=0.1, stop=65.0)

# Prior
plot( θ_range, x -> pdf(Normal(m0, sqrt(v0)), x), color="red", label="Prior", xlabel="θ", ylabel="p(θ)")

# Likelihood
plot!(θ_range, x -> pdf(message, x), color="blue", label="Likelihood")

# Posterior
plot!(θ_range, x -> pdf(posterior, x), color="purple", linestyle=:dash, label="Posterior", size=(800,300))
```

Out[34]:



The prior is quite wide, indicating the company has a lot of uncertainty about participants' response speeds. The likelihood is sharply peaked, even after only 4 questions. Note that the posterior is a weighted average of the prior- and likelihood-based messages. In this case, it is closer to the likelihood because the likelihood variance, 4, is much smaller than the prior variance 20.

Exercise

Suppose each question was timed by a different interviewer, and that the interviewers differ vastly in how precise they record response times. How can we incorporate this knowledge into the model?

In [ ]:

Probabilistic Programming 2: Bayesian regression and classification

Goal

- Learn how to infer a posterior distribution for a linear regression model using a probabilistic programming language.
- Learn how to infer a posterior distribution for a linear classification model using a probabilistic programming language.

Materials

- Mandatory
  - This notebook.
  - Lecture notes on regression.
  - Lecture notes on discriminative classification.
- Optional
  - Bayesian linear regression (Section 3.3 Bishop)
  - Bayesian logistic regression (Section 4.5 Bishop)
  - Cheatsheets: how does Julia differ from Matlab / Python.

In [1]:
```
using Pkg; Pkg.activate("../../."); Pkg.instantiate();
using IJulia; try IJulia.clear_output(); catch _ end
```

Out[1]: 0

Problem: Economic growth

In 2008, the credit crisis sparked a recession in the US, which spread to other countries in the ensuing years. It took most countries a couple of years to recover. Now, the year is 2011. The Turkish government is asking you to estimate whether Turkey is out of the recession. You decide to look at the data of the national stock exchange to see if there's a positive trend.

In [2]:
```
using CSV
using DataFrames
using LinearAlgebra
using Distributions
using StatsFuns
using RxInfer
using Plots
default(label="", margin=10Plots.pt)
```

Data

We are going to start with loading in a data set. We have daily measurements from Istanbul, from the 5th of January 2009 until 22nd of February 2011. The dataset comes from an online resource for machine learning data sets: the UCI ML Repository.

In [3]:
```
# Read CSV file
df = DataFrame(CSV.File("../datasets/stock_exchange.csv"))
```

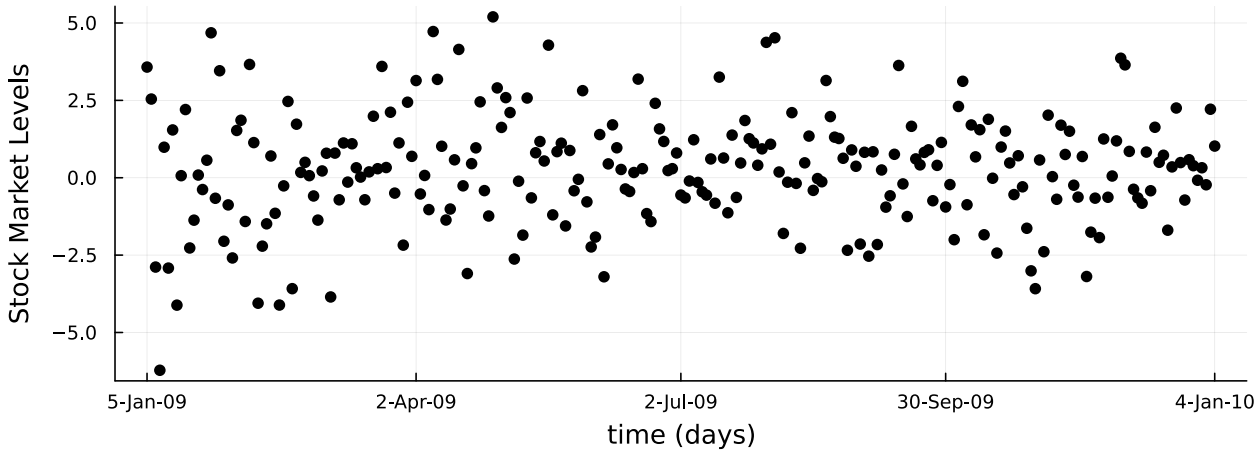
Out[3]:251×2 DataFrame *226 rows omitted*

Row	date	ISE
	String15	Float64
1	5-Jan-09	3.57537
2	6-Jan-09	2.54259
3	7-Jan-09	-2.88617
4	8-Jan-09	-6.22081
5	9-Jan-09	0.98599
6	12-Jan-09	-2.9191
7	13-Jan-09	1.54453
8	14-Jan-09	-4.11676
9	15-Jan-09	0.0661905
10	16-Jan-09	2.20373
11	19-Jan-09	-2.26925
12	20-Jan-09	-1.37087
13	21-Jan-09	0.0864697
⋮	⋮	⋮
240	17-Dec-09	-1.69489
241	18-Dec-09	0.350124
242	21-Dec-09	2.25302
243	22-Dec-09	0.48947
244	23-Dec-09	-0.721131
245	24-Dec-09	0.581665
246	25-Dec-09	0.389112
247	28-Dec-09	-0.0811768
248	29-Dec-09	0.322285
249	30-Dec-09	-0.227405
250	31-Dec-09	2.21384
251	4-Jan-10	1.02294

We can plot the evolution of the stock market values over time.

In [4]:
```
# Count number of samples
time_period = 1:251
num_samples = length(time_period)

# Extract columns
dates_num = 1:num_samples
dates_str = df[time_period,1]
stock_val = df[time_period,2]

# Set xticks
xtick_points = Int64.(round.(range(1, stop=num_samples, length=5)))

# Scatter exchange levels
scatter(dates_num,
        stock_val,
        color="black",
        label="",
        ylabel="Stock Market Levels",
        xlabel="time (days)",
        xticks=(xtick_points, [dates_str[i] for i in xtick_points]),
        size=(800,300))
```

Out[4]:



Model specification

We have a date $x_i \in \mathbb{R}$, referred to as a "covariate" or input variable, and the value of the stock exchange at that time point $y_i \in \mathbb{R}$, referred to as a "response" or output variable. A regression model has parameters θ, used to predict $y = (y_1, \ldots, y_N)$ from $x = (x_1, \ldots, x_N)$. We are looking for a posterior distribution for the parameters θ:

$$\underbrace{p(\theta \mid y, x)}_{\text{posterior}} \propto \underbrace{p(y \mid x, \theta)}_{\text{likelihood}} \cdot \underbrace{p(\theta)}_{\text{prior}}$$

We assume each observation y_i is generated via:

$$y_i = f_\theta(x_i) + e_i$$

where e_i is white noise, $e_i \sim \mathcal{N}(0, \sigma_y^2)$, and the regression function f_θ is linear: $f_\theta(x) = x\theta_1 + \theta_2$. The parameters consist of a slope coefficient θ_1 and an intercept θ_2, which are summarized into the vector $\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$. In practice, we augment the data point x with a 1, i.e., $\begin{bmatrix} x \\ 1 \end{bmatrix}$, so that we may define $f_\theta(x) = \theta^\top x$.

Likelihood

If we integrate out the noise e, then we obtain a Gaussian likelihood function centered on $f_\theta(x)$ with variance σ_Y^2:

$$p(y_i \mid x_i, \theta) = \mathcal{N}(y_i \mid f_\theta(x_i), \sigma_y^2) \ .$$

But this is just for a single sample and we have an entire data set. The likelihood of all (x, y) is:

$$
\begin{aligned}
p(y \mid x, \theta) &= \prod_{i=1}^{N} p(y_i \mid x_i, \theta) \\
&= \prod_{i=1}^{N} \mathcal{N}(y_i \mid f_\theta(x_i), \sigma_y^2)\,.
\end{aligned}
$$

Prior distribution

We know that the weights are real numbers and that they can be negative. That motivates us to use a Gaussian prior:

$$
p(\theta) = \mathcal{N}(\theta \mid \mu_\theta, \Sigma_\theta)\,.
$$

We can specify these equations almost directly in our PPL.

In [5]:
```
@model function linear_regression(y, X, μ_θ, Σ_θ, σ2, N)
    "Bayesian linear regression"

    # Prior distribution of coefficients
    θ ~ MvNormalMeanCovariance(μ_θ, Σ_θ)

    for i = 1:N

        # Likelihood of i-th sample
        y[i] ~ NormalMeanVariance(dot(θ,X[i]), σ2)

    end
end
```

In [6]:
```
# Prior parameters
μ_θ, Σ_θ = (zeros(2), diagm(ones(2)))

# Likelihood variance
σ2_y = var(stock_val)
```

Out[6]:
```
3.2657337986971937
```

Now that we have our model, it is time to infer parameters.

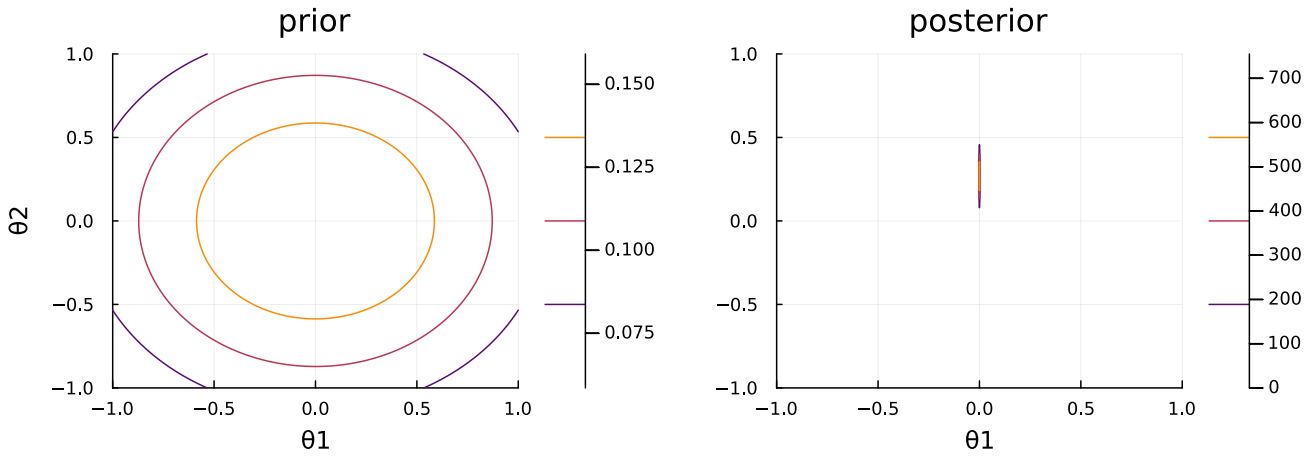
In [7]:
```
results = infer(
    model       = linear_regression(μ_θ=μ_θ, Σ_θ=Σ_θ, σ2=σ2_y, N=num_samples),
    data        = (y = stock_val, X = [[dates_num[i], 1.0] for i in 1:num_samples]),
)
```

Out[7]:
```
Inference results:
  Posteriors       | available for (θ)
```

Let's visualize the resulting posterior.

In [8]:
```
# Extract posterior weights
post_θ = results.posteriors[:θ]

# Define ranges for plot
x1 = range(-1., length=501, stop=1.)
x2 = range(-1., length=501, stop=1.)

# Draw contour plots of distributions
prior_θ = MvNormal(μ_θ, Σ_θ)
p1a = contour(x1, x2, (x1,x2) -> pdf(prior_θ, [x1,x2]), levels=3, xlabel="θ1", ylabel="θ2", title="prior")
p1b = contour(x1, x2, (x1,x2) -> pdf(post_θ, [x1,x2]), levels=3, xlabel="θ1", title="posterior")
plot(p1a, p1b, size=(900,300))
```

Out[8]:



It has become quite sharply peaked in a small area of parameter space.

We can extract the MAP point estimate to compute and visualize the most probable regression function f_θ.

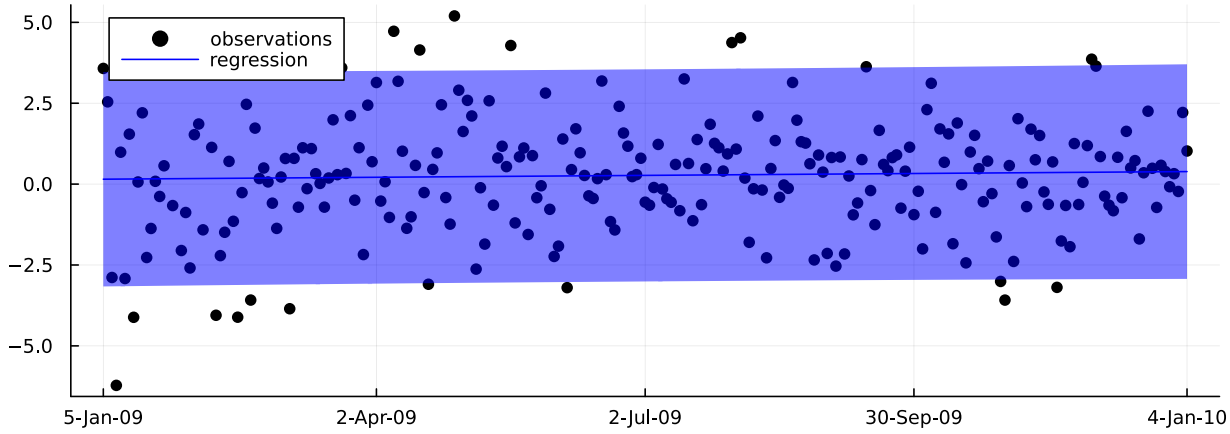
In [9]:
```
# Extract estimated weights
θ_MAP = mode(post_θ)
θ_Σ   = cov( post_θ)

# Report results
println("Slope coefficient = "*string(θ_MAP[1]))
println("Intercept coefficient = "*string(θ_MAP[2]))

# Make predictions
regression_estimated = zeros(num_samples)
regression_uncertainty = zeros(num_samples)
for (i,date) in enumerate(dates_num)
    regression_estimated[i] = date * θ_MAP[1] .+ θ_MAP[2];
    regression_uncertainty[i] = [date, 1.]' *θ_Σ * [date, 1.] + σ2_y
end
```

```
Slope coefficient = 0.000941327449846718
Intercept coefficient = 0.15081258570809555
```

Let's visualize the estimated regression function.

In [10]:
```
# Visualize observations
scatter(dates_num, stock_val, color="black", xticks=(xtick_points, [dates_str[i] for i in xtick_points]), l

# Overlay regression function
plot!(dates_num, regression_estimated, ribbon=regression_uncertainty, color="blue", label="regression", siz
```

Out[10]:



The slope coefficient θ_1 is nearly zero and the plot shows a horizontal line. So the ISE did not experienced a decline, but also did not grow in 2009.

Exercise

Change the `time period` variable. Re-run the regression and see how the results change.

Forecasting

We've answered the Turkish government's question. But before they change their country's economic policy, they want to know what the future looks like if the stock market continues on this trend. In other words, they want us to make predictions for future outputs given our current regression coefficient estimates.

We can request the toolbox to make predictions for future outputs by providing `missing` data points.

In [11]:
```
future = DataFrame(CSV.File("../datasets/stock_futures.csv"))
```

Out[11]:251×2 DataFrame *226 rows omitted*

Row	date	ISE
	String15	Missing
1	5-Jan-10	*missing*
2	6-Jan-10	*missing*
3	7-Jan-10	*missing*
4	8-Jan-10	*missing*
5	11-Jan-10	*missing*
6	12-Jan-10	*missing*
7	13-Jan-10	*missing*
8	14-Jan-10	*missing*
9	15-Jan-10	*missing*
10	18-Jan-10	*missing*
11	19-Jan-10	*missing*
12	20-Jan-10	*missing*
13	21-Jan-10	*missing*
⋮	⋮	⋮
240	21-Dec-10	*missing*
241	22-Dec-10	*missing*
242	23-Dec-10	*missing*
243	24-Dec-10	*missing*
244	27-Dec-10	*missing*
245	28-Dec-10	*missing*
246	29-Dec-10	*missing*
247	30-Dec-10	*missing*
248	31-Dec-10	*missing*
249	3-Jan-11	*missing*
250	4-Jan-11	*missing*
251	5-Jan-11	*missing*

In [12]:
```
all_N = num_samples + length(future[!,:date])
all_d = [df[!,:date]; future[!,:date]]
all_y = [stock_val; future[!,:ISE]]
all_X = [[i, 1.0] for i in 1:all_N]

results = infer(
    model       = linear_regression(μ_θ=μ_θ, Σ_θ=Σ_θ, σ2=σ2_y, N=all_N),
    data        = (y = all_y, X = all_X,),
    predictvars = (y = KeepLast(),),
)
```

Out[12]:
```
Inference results:
  Posteriors       | available for (θ)
  Predictions      | available for (y)
```

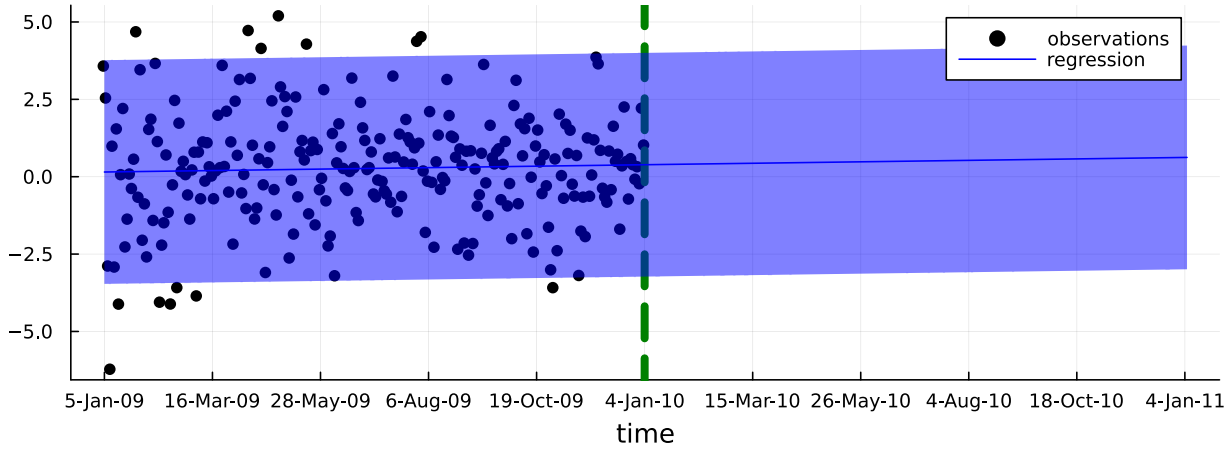
In [13]:
```
regression_estimated = mode.(results.predictions[:y])
regression_uncertainty = 2std.(results.predictions[:y]) # 2 standard deviations

final_prediction = regression_estimated[end]
final_uncertainty = regression_uncertainty[end]
println("Final predicted stock value = $final_prediction (±$final_uncertainty)")
```

```
Final predicted stock value = 0.6233589655311479 (±3.6142682793047856)
```

Although the prediction itself is positive, its uncertainty tells us that we should not trust this result too much. Let's see what this looks like in a plot.

In [14]:
```
# Visualize observations
plot(xlabel="time", xticks=(1:50:all_N, [all_d[i] for i in 1:50:all_N]), size=(800,300))
scatter!(dates_str, stock_val, color="black", label="observations")
vline!([251], color="green", linewidth=5, linestyle=:dash)

# Overlay regression function
plot!(1:all_N, regression_estimated, ribbon=regression_uncertainty, color="blue", label="regression")
```

Out[14]:



The ribbon is relatively wide. This tells us that the future looks uncertain for the stock market.

Problem: Medical Diagnosis

A company is trying to develop a measurement device that tells a patient whether they suffer from Chronic Obstructive Pulmonary Disease (COPD). They believe they can detect certain compounds in a saliva sample that indicate the presence of COPD. To test the device, they collect data from both COPD patients and healthy controls. They train a classifier and make predictions on a test sample. If the diagnosis can be accurately predicted, then the new device is deemed an informative tool and will be brought to market.

Data

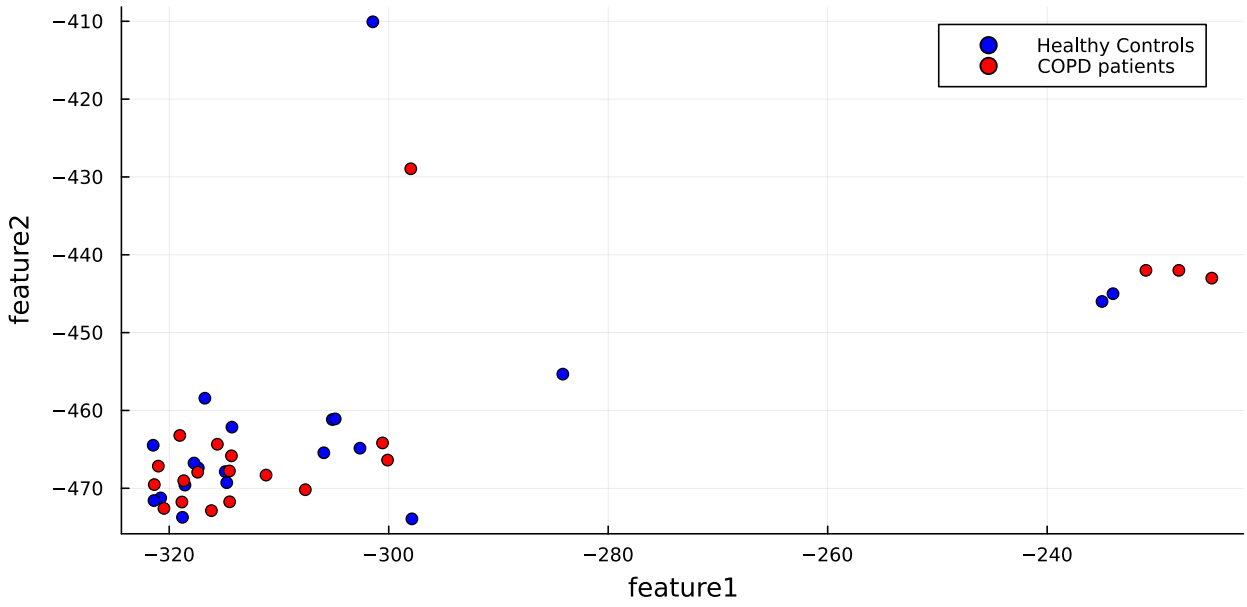
The data set comes from the UCI ML Repository. It contains measurements of 79 participants, split into 40 samples for training and 39 for testing. The columns in the data file marked *:x* are biometric features (x1,x2 = measured signal, x3 = gender, x4 = age, x5 = smoking) and the final column marked *:y* is the diagnosis (healthy control =0, COPD =1).

In [15]:
```
# Read CSV file
train_data = DataFrame(CSV.File("../datasets/diagnosis_train.csv"))

# Split dataframe into features and labels
features_train = Matrix(train_data[:,1:5])
labels_train = Vector(train_data[:,6])

# Store number of features
num_features = size(features_train,2)

# Number of training samples
num_train = size(features_train,1);
```

Let's have a look at measurements from the device.

In [16]:
```
plot(xlabel="feature1", ylabel="feature2", size=(800,400))
scatter!(features_train[labels_train .== 0, 1], features_train[labels_train .== 0, 2], color="blue", label=
scatter!(features_train[labels_train .== 1, 1], features_train[labels_train .== 1, 2], color="red",  label=
```

Out[16]:



Mmhh, it doesn't look like the samples can easily be separated. Are these measurements really informative?

Exercise

The plot above shows features 1 and 2. Have a look at the other combinations of features.

Model specification

We have features X, labels Y and parameters θ. Same as with regression, we are looking for a posterior distribution of the classification parameters:

$$\underbrace{p(\theta \mid Y, X)}_{\text{posterior}} \propto \underbrace{p(Y \mid X, \theta)}_{\text{likelihood}} \cdot \underbrace{p(\theta)}_{\text{prior}}$$

The likelihood in this case will be of a probit form:

$$p(Y \mid X, \theta) = \prod_{i=1}^{N} \mathcal{B}\big(Y_i \mid \Phi(f_\theta(X_i))\big)\,.$$

As you can see it is a Bernoulli distribution with a cumulative normal distribution as transfer (a.k.a. *link*) function:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{t^2}{2}\right) \mathrm{d}t\,.$$

The transfer function maps the input ($f_\theta(X_i)$) to the interval $(0, 1)$ so that the result acts as a rate parameter to the Bernoulli. Check Bert's lecture on discriminative classification for more information.

We will use a Gaussian prior distribution for the classification parameters θ:

$$p(\theta) = \mathcal{N}(\theta \mid \mu_\theta, \Sigma_\theta)\,.$$

You have probably noticed that this combination of likelihood and prior is not part of the family of conjugate pairings. As such, we don't have an exact posterior. The Laplace approximation is one procedure but under the hood, our toolbox is actually performing a different procedure for obtaining the posterior parameters: moment matching). The cumulative normal distribution allows for integrating the product of prior and likelihood by hand, with respect to the first (mean) and second (variance) moments. The toolbox is essentially performing a lookup for the analytically derived formula, which computationally cheaper than performing the iterative steps necessary for the Laplace approximation.

In [17]:
```
# Parameters for priors
μ_θ = zeros(num_features+1,)
Σ_θ = diagm(ones(num_features+1));
```

In [18]:
```
@model function linear_classification(y,X, μ_θ, Σ_θ, N)
    "Bayesian classification model"
```

```
        # Weight prior distribution
        θ ~ MvNormalMeanCovariance(μ_θ, Σ_θ)

        # Binary likelihood
        for i = 1:N

            y[i] ~ Probit(dot(θ, X[i]))

        end
    end
```

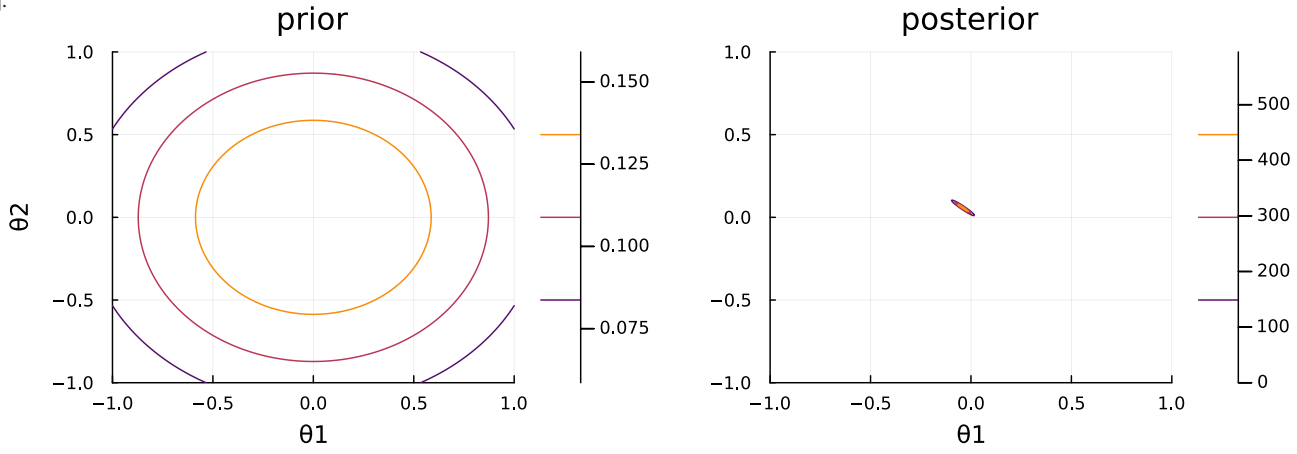
In [19]:
```
results = infer(
    model       = linear_classification(μ_θ=μ_θ, Σ_θ=Σ_θ, N=num_train),
    data        = (y = labels_train, X = [[features_train[i,:]; 1.0] for i in 1:num_train]),
    returnvars  = (θ = KeepLast()),
    iterations  = 10,
)
```

Out[19]:
```
Inference results:
  Posteriors       | available for (θ)
```

Unfortunately, we cannot visualize a distribution of more than 2 dimensions. But we can visualize a pair of dimensions:

In [20]:
```
# Coefficients to visualize
cix = [1,2]

# Reduce posterior distribution to chosen dimensions
m_cix = mean(results.posteriors[:θ])[cix]
S_cix = cov( results.posteriors[:θ])[cix,cix]
post_θ = MvNormal(m_cix, S_cix)

# Reduce prior distribution to chosen dimensions
prior_θ = MvNormal(μ_θ[cix], Σ_θ[cix,cix])

# Define ranges for plot
x1 = range(-1., length=500, stop=1.)
x2 = range(-1., length=500, stop=1.)

# Draw contour plots of distributions
p1a = contour(x1, x2, (x1,x2) -> pdf(prior_θ, [x1,x2]), levels=3, xlabel="θ1", ylabel="θ2", title="prior",
p1b = contour(x1, x2, (x1,x2) -> pdf(post_θ, [x1,x2]), levels=3, xlabel="θ1", title="posterior", label="")
plot(p1a, p1b, size=(900,300))
```

Out[20]:



Predict test data

The device should make accurate predictions for future patients. We can evaluate this with a test data set.

In [21]:
```
# Read CSV file
test_data = DataFrame(CSV.File("../datasets/diagnosis_test.csv"))

# Split dataframe into features and labels
features_test = Matrix(test_data[:,1:5])
labels_test = Vector(test_data[:,6])

# Number of test samples
num_test = size(features_test,1);
```

We can generate the most probable class labels by extracting the most probable classifier weights, calculating the dot product with the test feature vectors, and applying the Probit node's link function. This produces the class label probability and by rounding we get hard assignments to 0 or 1. This can be checked against the true test labels.

In [22]:
```
# Extract MAP estimate of classification parameters
θ_MAP = mode(results.posteriors[:θ])

# Compute dot product between parameters and test data
fθ_pred = [features_test ones(num_test,)] * θ_MAP

# Predict labels through probit
labels_pred = round.(normcdf.(fθ_pred));

# Compute classification accuracy of test data
accuracy_test = mean(labels_test .== labels_pred)

# Report result
println("Test Accuracy = "*string(accuracy_test*100)*"%")
```

```
Test Accuracy = 92.3076923076923%
```

Those predictions are very accurate. So, is the device informative after all?

Exercise

Re-run the classifier on just the saliva measurement features. Does the accuracy drop? And if so, should the device be used?

Probabilistic Programming 3: variational Bayes

Goal

- Know what is required to set up a variational Bayesian inference procedure.
- Learn how to infer the parameters of a Gaussian mixture model using a variational inference procedure.

Materials

- Mandatory
  - This notebook
  - Lecture notes on latent variable models
- Optional
  - Review of latent variable models
  - Differences between Julia and Matlab / Python.

Note that none of the material below is new. The point of the Probabilistic Programming sessions is to solve practical problems so that the concepts from Bert's lectures become less abstract.

In [1]:
```
using Pkg; Pkg.activate("../../."); Pkg.instantiate();
using IJulia; try IJulia.clear_output(); catch _ end
```

Out[1]:0

In [2]:
```
using Random
using JLD
using Statistics
using LinearAlgebra
using Distributions
using RxInfer
using Optimisers
using ColorSchemes
using LaTeXStrings
using Plots
default(label="", linewidth=3, margin=10Plots.pt)
include("../scripts/clusters.jl");
```

Problem: Stone Tools

Archeologists have asked for your help in analyzing data on tools made of stone. It is believed that primitive humans created tools by striking stones with others. During this process, the stone loses flakes, which have been preserved. The archeologists have recovered these flakes from various locations and time periods and want to know whether this stone tool shaping process has improved over the centuries.

Data

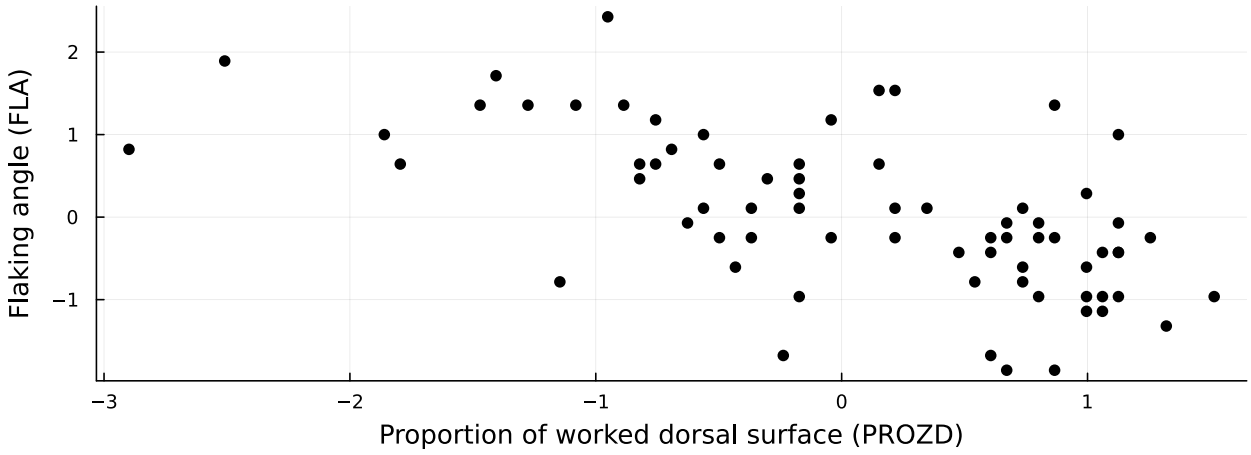
The data is available from the UCI Machine Learning Repository. Each instance represents summary information of the stone flakes for a particular site. We will be using the attributes *flaking angle* (FLA) and the *proportion of the dorsal surface worked* (PROZD) for now.

In [3]:
```
data = load("../datasets/stone_flakes.jld");
```

I've done some pre-processing on the data set, namely z-scoring and removing two outliers. This reduces the scale of the attributes which helps numerical stability during optimization. Now let's visualize the data with a scatterplot.

In [4]:
```
scatter(data["observations"][:,1],
        data["observations"][:,2],
        color="black",
        xlabel="Proportion of worked dorsal surface (PROZD)",
        ylabel="Flaking angle (FLA)",
        size=(800,300))
```

Out[4]:



Model specification

We will be clustering this data with a Gaussian mixture model, to see if we can identify clear types of stone tools. The generative model for a Gaussian mixture consists of:

$$p(X, z, \pi, \mu, \Lambda) = \underbrace{p(X \mid z, \mu, \Lambda)}_{\text{likelihood}} \times \underbrace{p(z \mid \pi)}_{\text{prior latent variables}} \times \underbrace{p(\mu \mid \Lambda)\, p(\Lambda)\, p(\pi)}_{\text{prior parameters}}$$

with the likelihood of observation X_i being a Gaussian raised to the power of the latent assignment variables z

$$p(X_i \mid z, \mu, \Lambda) = \prod_{k=1}^{K} \mathcal{N}(X_i \mid \mu_k, \Lambda_k^{-1})^{z_i = k}$$

the prior for each latent variable z_i being a Categorical distribution

$$p(z_i \mid \pi) = \text{Categorical}(z_i \mid \pi)$$

and priors for the parameters being

$$\begin{aligned} p(\Lambda_k) &= \text{Wishart}(\Lambda_k \mid V_0, n_0) && \text{for all } k, \\ p(\mu_k \mid \Lambda_k) &= \mathcal{N}(\mu_k \mid m_0, \Lambda_k^{-1}) && \text{for all } k, \\ p(\pi) &= \text{Dirichlet}(\pi \mid a_0) \quad . \end{aligned}$$

We can implement these equations nearly directly in ReactiveMP.

In [5]:
```
# Data dimensionality
num_features = size(data["observations"],2)

# Sample size
num_samples = size(data["observations"],1)
```

```
# Number of mixture components
num_components = 3;
```

Mixture models can be sensitive to initialization, so we are going to specify the prior parameters explicitly.

In [6]:
```
# Prior scale matrices
V0 = repeat(diageye(num_features), 1,1,3)

# Prior degrees of freedom
n0 = num_features

# Prior means
m0 = [ 1.0 0.0 -1.0;
       -1.0 0.0  1.0];

# Prior concentration parameters
a0 = ones(num_components);
```

Now to specify the full model.

In [7]:
```
@model function GMM(X, n0, V0, m0, a0, K, N)
    "Bayesian Gaussian mixture model"

    local μ
    local Λ

    # Component parameters
    for k in 1:K
        Λ[k] ~ Wishart(n0, V0[:,:,k])
        μ[k] ~ MvNormalMeanPrecision(m0[:,k], Λ[k])
    end

    # Mixture weights
    π ~ Dirichlet(a0)

    for i in 1:N

        # Latent assignment variable
        z[i] ~ Categorical(π)

        # Mixture distribution
        X[i] ~ NormalMixture(switch = z[i], m = μ, p = Λ)

    end
end
```

Set up the inference procedure.

In [8]:
```
# Map data to list of vectors
observations = [data["observations"][i,:] for i in 1:num_samples]

# Initialize variational distributions
init = @initialization begin
    q(π) = Dirichlet(a0)
    q(μ) = [MvNormalMeanCovariance(m0[:,k], diageye(num_features)) for k in 1:num_components]
    q(Λ) = [Wishart(n0, V0[:,:,k]) for k in 1:num_components]
end

# Iterations of variational inference
num_iters = 100

# Variational inference
results = infer(
    model          = GMM(n0=n0, V0=V0, m0=m0, a0=a0, K=num_components, N=num_samples),
    data           = (X = observations,),
    constraints    = MeanField(),
    initialization = init,
    iterations     = num_iters,
    free_energy    = true,
    showprogress   = true,
)
```

```
Progress: 100%|████████████████████████████████████| Time: 0:00:03
```

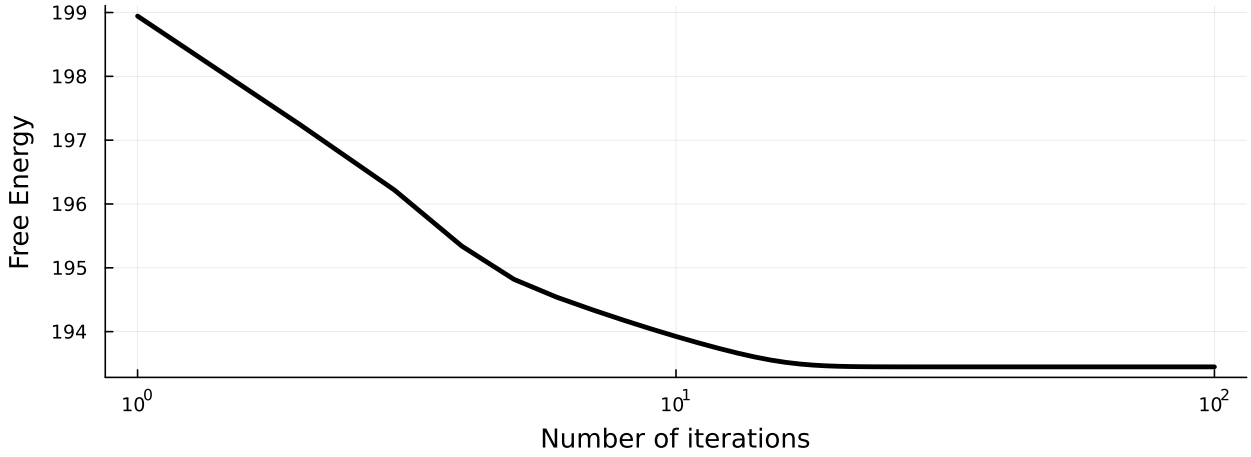
Out[8]:
```
Inference results:
  Posteriors       | available for (π, μ, Λ, z)
  Free Energy:     | Real[198.944, 197.247, 196.218, 195.341, 194.82, 194.541, 194.341, 194.178, 194.041, 1
93.924  …  193.448, 193.448, 193.448, 193.448, 193.448, 193.448, 193.448, 193.448, 193.448, 193.448]
```

Alright, we're done. Let's track the evolution of free energy over iterations.

In [9]:

```
plot(1:num_iters,
     results.free_energy,
     color="black",
     xscale=:log10,
     xlabel="Number of iterations",
     ylabel="Free Energy",
     size=(800,300))
```

Out[9]:

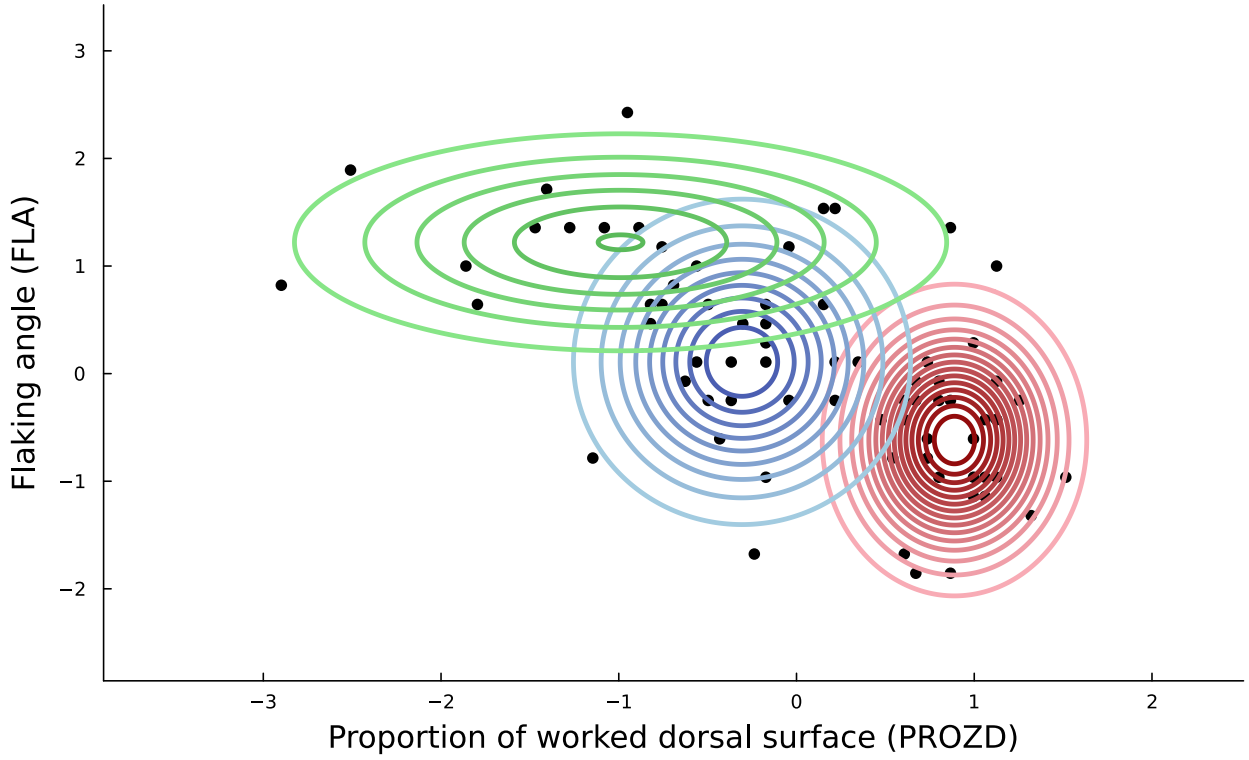


That looks like it is nicely decreasing. Let's now visualize the cluster on top of the observations.

In [10]:

```
# Extract mixture weights
π_hat = mean(results.posteriors[:π][num_iters])

# Extract component means
μ_hat = mean.(results.posteriors[:μ][num_iters])

# Extract covariance matrices
Σ_hat = inv.(mean.(results.posteriors[:Λ][num_iters]))

# Select dimensions to plot
xlims = [minimum(data["observations"][:,1])-1, maximum(data["observations"][:,1])+1]
ylims = [minimum(data["observations"][:,2])-1, maximum(data["observations"][:,2])+1]

# Plot data and overlay estimated posterior probabilities
plot_clusters(data["observations"][:, 1:2],
              μ=μ_hat,
              Σ=Σ_hat,
              xlims=xlims,
              ylims=ylims,
              xlabel="Proportion of worked dorsal surface (PROZD)",
              ylabel="Flaking angle (FLA)",
              colors=[:reds, :blues, :greens],
              figsize=(800,500))
```

Out[10]:



That doesn't look bad. The three Gaussians nicely cover all samples.

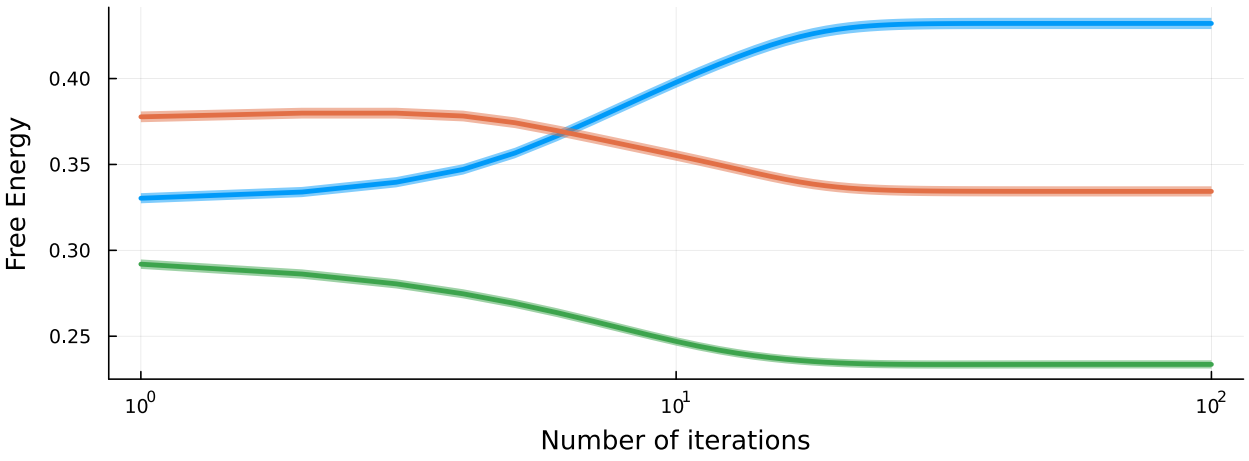
Exercise

Play around with the number of components. Can you get an equally good coverage with just 2 components? What if you had 4?

We can also plot the evolution of the parameters over iterations of the variational inference procedure.

In [11]:
```
# Extract mean and standard deviation from
mean_π_iters = cat(mean.(results.posteriors[:π])..., dims=2)
vars_π_iters = cat(var.( results.posteriors[:π])..., dims=2)

plot(1:num_iters,
     mean_π_iters',
     ribbon=vars_π_iters',
     xscale=:log10,
     xlabel="Number of iterations",
     ylabel="Free Energy",
     size=(800,300))
```

Out[11]:



Exercise

Plot the evolution of one of the component means over iterations of variational inference, including its variance.

Problem: robot arm localization

A manufacturing company has a robot arm with servo motors that rotate the joints. The engineers at the company want a probabilistic description of the location of the arm in the world frame (i.e., Cartesian coordinates). They want to be able to do this from the measured angle at the joint. This is an example of a Cartesian to polar coordinate transformation, described by the following function:

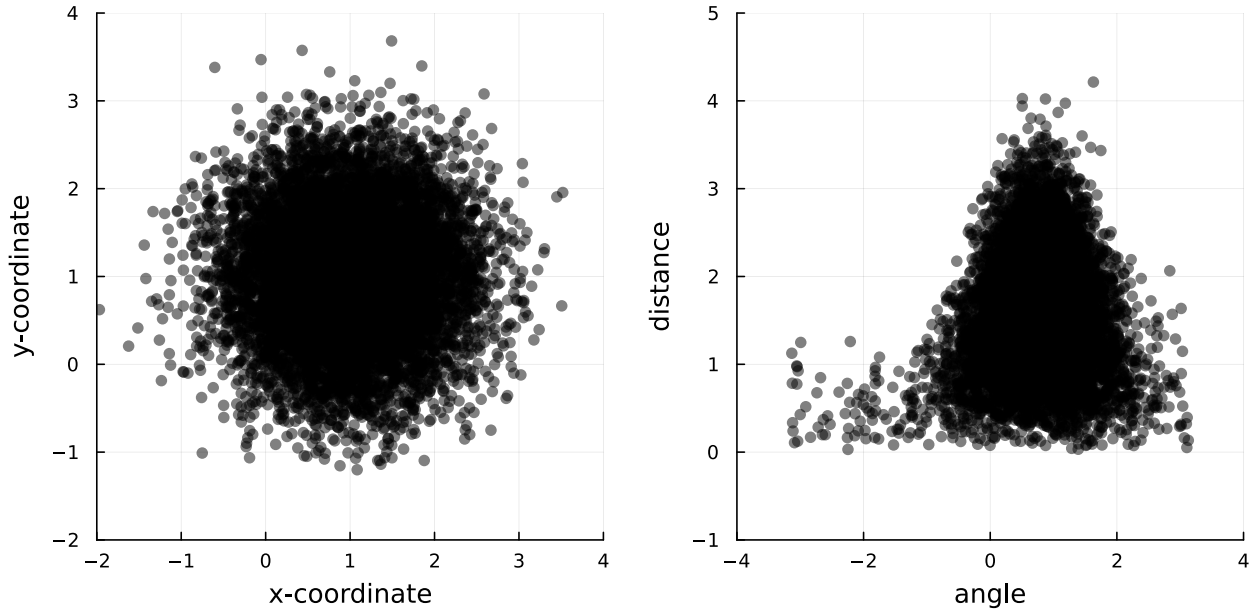
In [12]:
```
f(x) = [atan(x[2],x[1]);
        sqrt(x[1]^2 + x[2]^2)]
```

Out[12]:`f (generic function with 1 method)`

Nonlinear transformations of random variables are challenging, because they can drastically change the shape of a probability distribution. For example, suppose we have a random variable x that is Gaussian distributed: $x \sim \mathcal{N}(x|\mu, \Sigma)$. The new random variable $y = f(x)$ will not be Gaussian distributed. We can visualize this by sampling from x and applying the transformation to each sample.

In [13]:
```
D = 2
μ = ones(D)
Σ = 1/2 *diageye(D)

N = 10_000
x_samples = rand(MvNormal(μ,Σ),N)
y_samples = hcat([f(x_samples[:,i]) for i in 1:N]...)

p1 = scatter(x_samples[1,:],
             x_samples[2,:],
             color="black",
             alpha=0.5,
             xlabel="x-coordinate",
             ylabel="y-coordinate",
             xlims=(-2.,4.),
             ylims=(-2.,4.))

p2 = scatter(y_samples[1,:],
             y_samples[2,:],
             color="black",
             alpha=0.5,
             xlabel="angle",
             ylabel="distance",
             xlims=(-4., 4.),
             ylims=(-1,5.))

plot(p1,p2, layout=(1,2), size=(800,400))
```

Out[13]:



The distribution of the transformed samples looks like a triangle, which is different than the elliptic shape of a Gaussian distribution.

When models have nonlinear operations, we are often faced with a computation-vs-accuracy trade-off. We can spend a lot of time and effort to obtain an accurate description of the transformed random variable or we try to obtain a fast, cheap approximation. For example, we could

approximate the transformed variable y with a Gaussian distribution. This is not exactly the same as variational Bayesian inference, because it does not necessarily employ a prior distribution and a likelihood function. But it is similar in nature in that we want to optimize an approximation with a pre-specified alternative distribution.

Suppose the manufacturing company provides you with an angle measurement of $0.5\pi\left(\pm 1.0\right)$ and a distance measurement of $0.5\left(\pm 1e-3\right)$. We can infer a Gaussian distribution over the 2-dimensional coordinates. In this case, we also know the inverse function, so we can test against that.

In [14]:
```
@model function demo_nonlinear(z, μx,Σx,Σz)

    x ~ MvNormalMeanCovariance(μx,Σx)
    y := f(x)
    z ~ MvNormalMeanCovariance(y,Σz)

end
```

To incorporate the known nonlinear function as a factor node in the model, we must add a "meta" specification that selects the type of approximation. There are a few options for nonlinear functions:

- `Linearization`: the toolbox will automatically perform a first-order Taylor series approximation. This is a general technique, but suffers from a lack of accuracy.
- `Unscented`: this refers to the unscented transform, a popular technique that approximates the result of the transformation with a Gaussian distribution.
- `CVI` (Conjugate-computational Variational Inference): this is an advanced technique that uses both variational inference and importance sampling to approximate the transformed variable.

In [15]:
```
demo_meta = @meta begin
    # f() -> Linearization()
    # f() -> Unscented()
    f() -> CVI(MersenneTwister(256), 1000, 10, Optimisers.Adam())
end

μx = zeros(D)
Σx = diageye(D)
Σz = [1.0  0.0;
      0.0  1e-3]

inits = @initialization begin
    q(x) = MvNormalMeanCovariance(μx,Σx)
    μ(x) = MvNormalMeanCovariance(μx,Σx)
end

z_i = [0.5π, 0.5]

results = infer(
    model          = demo_nonlinear(μx=μx, Σx=Σx, Σz=Σz),
    data           = (z = z_i,),
    meta           = demo_meta,
    initialization = inits,
    iterations     = 10,
    returnvars     = (x = KeepLast(),),
    showprogress   = true,
)
```

```
Progress: 100%|████████████████████████████████████| Time: 0:00:02
```

Out[15]:
```
Inference results:
  Posteriors       | available for (x)
```

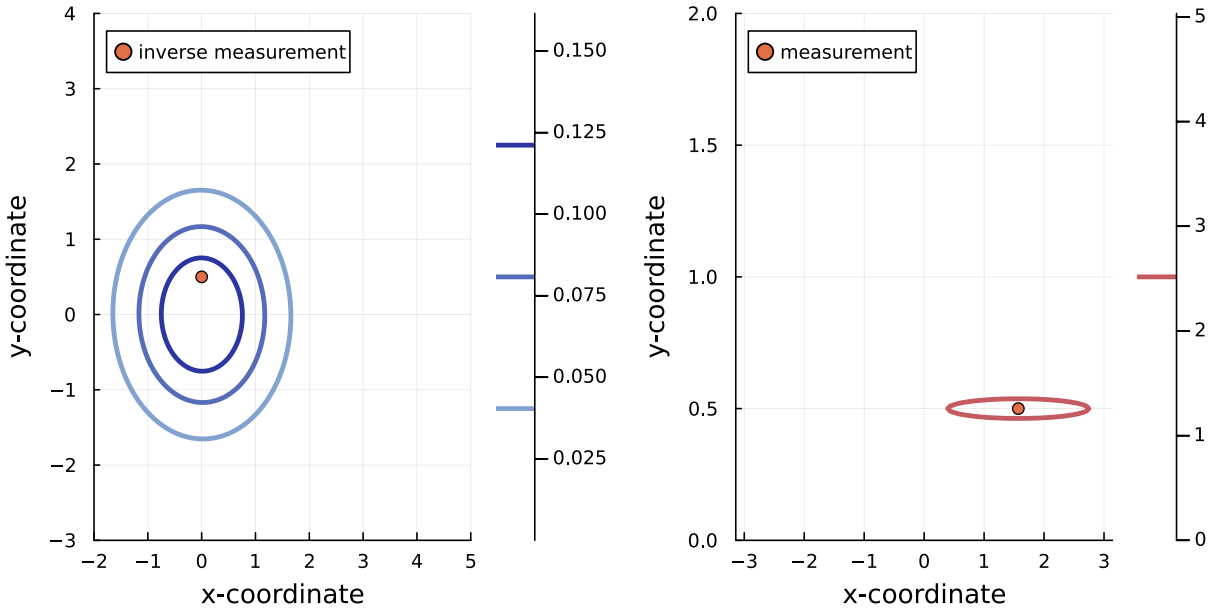
In [16]:
```
# Polar to Cartesian transformation
f_inv(y) = [y[2]*cos(y[1]),
            y[2]*sin(y[1])]

# Cartesian coordinate of point
x_i = f_inv(z_i)
```

Out[16]:
```
2-element Vector{Float64}:
 3.061616997868383e-17
 0.5
```

In [17]:
```
px = convert(MvNormalMeanCovariance, results.posteriors[:x])
p1 = contour(range(-2,5, length=201),
             range(-3,4, length=201),
             (x,y) -> pdf(px, [x,y]),
             color=:blues,
             levels = 3,
             xlabel="x-coordinate",
             ylabel="y-coordinate",
```

```
            xlims=(-2.,5.),
            ylims=(-3.,4.),
        )
scatter!([x_i[1]], [x_i[2]], label="inverse measurement")

pz_i = MvNormalMeanCovariance(z_i, Σz)
p2 = contour(range(-π, π, length=201),
            range( 0,2, length=201),
            (x,y) -> pdf(pz_i, [x,y]),
            color=:reds,
            levels = 1,
            xlabel="x-coordinate",
            ylabel="y-coordinate",
            xlims=(-π,π),
            ylims=( 0.,2.)
        )
scatter!([z_i[1]], [z_i[2]], label="measurement")

plot(p1,p2, layout=(1,2), size=(800,400))
```

Out[17]:



We have produced a Gaussian approximation of the inverse of the nonlinear function.

Exercise

Try out some of the other approximations for various angle and distance measurements. What works and what doesn't?

Probabilistic Programming 4: Bayesian filtering & smoothing

Goal

- Learn how to setup problems for dynamical systems.
- Learn how to infer states and noise in linear Gaussian state-space model using variational inference.

Materials

- Mandatory
  - This notebook
  - Lecture notes on dynamical models
- Optional
  - Review of latent variable models
  - Bayesian Filtering & Smoothing
  - Differences between Julia and Matlab / Python.

Note that none of the material below is new. The point of the Probabilistic Programming sessions is to solve practical problems so that the concepts from Bert's lectures become less abstract.

In [1]:
```
using Pkg; Pkg.activate("../../."); Pkg.instantiate();
using IJulia; try IJulia.clear_output(); catch _ end
```

Out[1]:0

In [2]:
```
using JLD
using Statistics
using LinearAlgebra
using Distributions
using RxInfer
using ColorSchemes
using LaTeXStrings
using Plots
default(label="", grid=false, linewidth=3, margin=10Plots.pt)
```

Problem: shaking buildings

Suppose you are contacted to estimate how resistant a building is to shaking caused by minor earthquakes. You decide to model the building as a mass-spring-damper system, described by:

$$m\ddot{x} + c\dot{x} + kx = w\,,$$

where m corresponds to the mass of the building, c is friction and k the stiffness of the building's main supports. You don't know the external force acting upon the building and decide to model it as white noise w. In essence, this means you think the building will be pushed to the left as strongly on average as it will be pushed to the right. A simple discretization scheme with substituted variable $z = \begin{bmatrix} x & \dot{x} \end{bmatrix}$ and time-step Δt yields:

$$z_k = \underbrace{\begin{bmatrix} 1 & \Delta t \\ \frac{-k}{m}\Delta t & \frac{-c}{m}\Delta t + 1 \end{bmatrix}}_{A} z_{k-1} + q_k\,,$$

where $q_k \sim \mathcal{N}(0, Q)$ with covariance matrix Q.

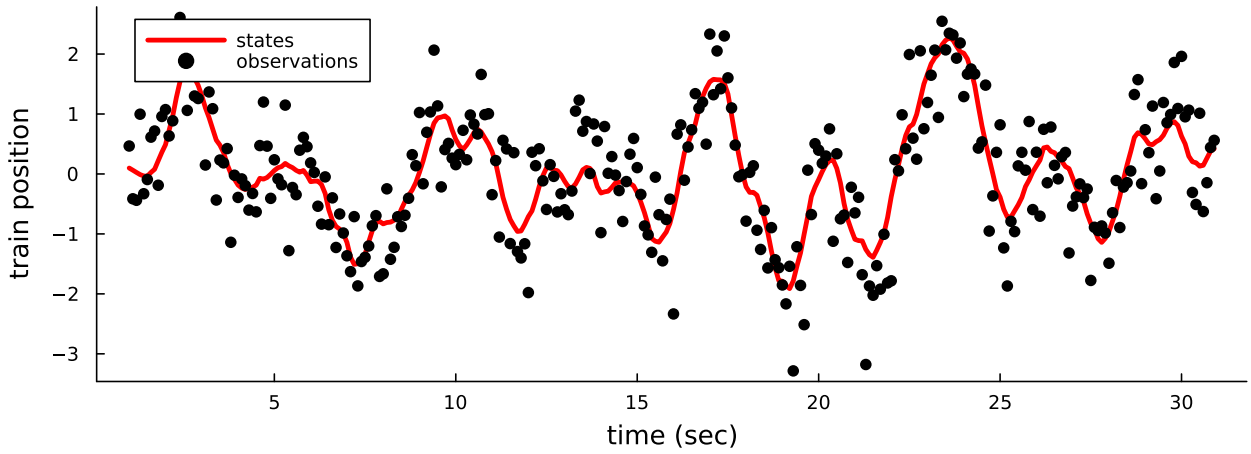
You place a series of sensors on the building that measure the displacement:

$$y_k = \underbrace{\begin{bmatrix} 1 & 0 \end{bmatrix}}_{C} z_k + r_k$$

where the measurement noise is white as well: $r_k \sim \mathcal{N}(0, \sigma^2)$. You have a good estimate of the variance σ^2 from previous sensor calibrations and decide to consider it a known variable.

In [3]:
```
# Load data from file
data = load("../datasets/shaking_buildings.jld")

# Data
states = data["states"]
observations = data["observations"]

# Parameters
mass = data["m"]
friction = data["c"]
stiffness = data["k"]

# Measurement noise variance
σ = data["σ"]

# Time
Δt = data["Δt"]
T = length(observations)
time = range(1,step=Δt,length=T)

plot(time, states[1,:], color="red", label="states", xlabel="time (sec)", ylabel="train position")
scatter!(time, observations, color="black", label="observations", legend=:topleft, size=(800,300))
```

Out[3]:



Model specification

Following the steps from the lecture on Dynamic Systems, we can derive the following probabilistic state-space model:

$$p(z_k \mid z_{k-1}) = \mathcal{N}(z_k \mid Az_{k-1}, Q)$$
$$p(y_k \mid z_k) = \mathcal{N}(y_k \mid Cz_k, \sigma^2)\,.$$

For now, we will use a simple structure for the process noise covariance matrix, e.g. $Q = I$. If we consider a Gaussian prior distribution for the initial state

$$p(z_0) = \mathcal{N}(m_0, S_0)\,,$$

we obtain a complete generative model:

$$\underbrace{p(y_{1:T}, z_{0:T})}_{\text{generative model}} = \underbrace{p(z_0)}_{\text{prior}} \prod_{k=1}^{T} \underbrace{p(y_k \mid z_k)}_{\text{likelihood}} \underbrace{p(z_k \mid z_{k-1})}_{\text{state transition}}$$

To define this model in RxInfer, we must start with the process matrices:

In [4]:
```
# Transition matrix
A = [1 Δt; -stiffness/mass*Δt -friction/mass*Δt+1]

# Emission matrix
C = [1.0, 0.0]

# Set process noise covariance matrix
Q = diagm(ones(2))
```

Out[4]:
```
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

Next, we define a linear Gaussian dynamical system with only the states as unknown variables:

In [5]:
```
@model function LGDS(y, A,C,Q, σ, T)
    "State estimation in linear Gaussian dynamical system"

    # Prior state
    z_0 ~ MvNormalMeanCovariance(zeros(2), diageye(2))

    z_kmin1 = z_0
    for k in 1:T

        # State transition
        z[k] ~ MvNormalMeanCovariance(A * z_kmin1, Q)

        # Likelihood
        y[k] ~ NormalMeanVariance(dot(C, z[k]), σ)

        # Update recursive aux
        z_kmin1 = z[k]

    end
end
```

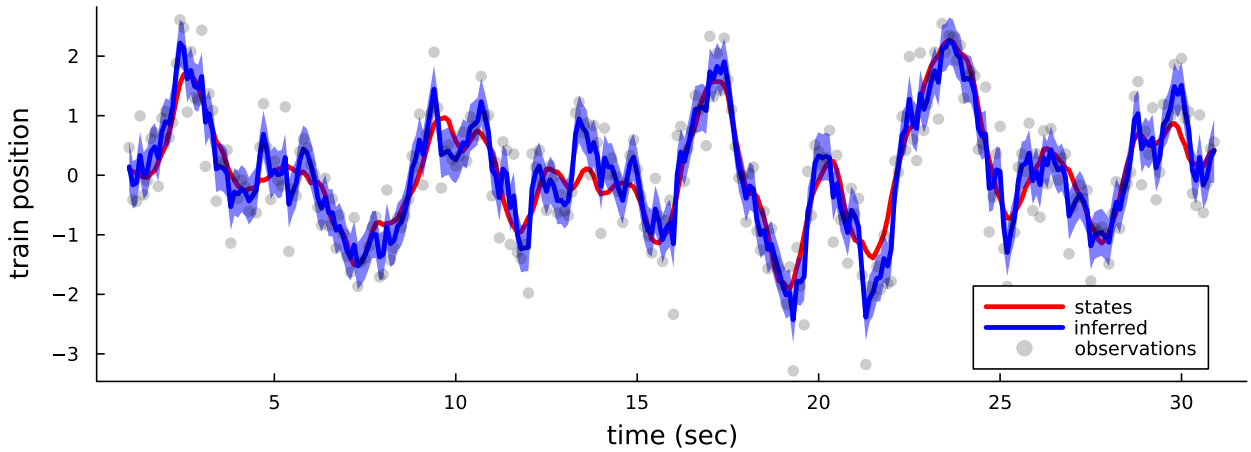
```
results = infer(
    model       = LGDS(A=A,C=C,Q=Q, σ=σ, T=T),
    data        = (y = [observations[k] for k in 1:T],),
    free_energy = true,
)
```

Out[5]:
```
Inference results:
  Posteriors       | available for (z_0, z)
  Free Energy:     | Real[444.922]
```

Let's extract the inferred states and visualize them.

In [6]:
```
m_z = cat(mean.(results.posteriors[:z])...,dims=2)
v_z = cat(var.( results.posteriors[:z])...,dims=2)

plot(time, states[1,:], color="red", label="states", xlabel="time (sec)", ylabel="train position")
plot!(time, m_z[1,:], color="blue", ribbon=v_z[1,:], label="inferred")
scatter!(time, observations, color="black", alpha=0.2, label="observations", legend=:bottomright, size=(800,
```

Out[6]:



Mmmh... the inferred states are not smooth at all. This is most likely due to our process noise covariance matrix not being calibrated. So can we improve?

Of course, as Bayesians, we can just treat Q as an unknown random variable and infer its posterior distribution. Adjusting the model is straightforward. The probabilistic state-space model becomes:

$$\begin{aligned} p(z_k \mid z_{k-1}, Q) &= \mathcal{N}(z_k \mid A z_{k-1}, Q) \\ p(y_k \mid z_k) &= \mathcal{N}(y_k \mid C z_k, \sigma^2)\,, \end{aligned}$$

with priors

$$\begin{aligned} p(Q) &= \mathcal{W}^{-1}(\nu, \Lambda) \\ p(z_0) &= \mathcal{N}(m_0, S_0)\,. \end{aligned}$$

The $\mathcal{W}^{-1}$ represents an inverse-Wishart distribution with degrees-of-freedom ν and scale matrix Λ. This will give the following generative model:

$$\underbrace{p(y_{1:T}, z_{0:T}, Q)}_{\text{generative model}} = \underbrace{p(z_0)p(Q)}_{\text{priors}} \prod_{k=1}^{T} \underbrace{p(y_k \mid z_k)}_{\text{likelihood}} \underbrace{p(z_k \mid z_{k-1}, Q)}_{\text{state transition}}$$

Our model definition in ReactiveMP is only slightly larger:

In [7]:
```
@model function LGDS_Q(y, A,C,σ,T)
    "State estimation in a linear Gaussian dynamical system with unknown process noise"

    # Prior state
    z_0 ~ MvNormalMeanCovariance(zeros(2), diageye(2))

    # Process noise covariance matrix
    Q ~ InverseWishart(10, diageye(2))
```

```
        z_kmin1 = z_0
        for k in 1:T

            # State transition
            z[k] ~ MvNormalMeanCovariance(A * z_kmin1, Q)

            # Likelihood
            y[k] ~ NormalMeanVariance(dot(C, z[k]), σ^2)

            # Update recursive aux
            z_kmin1 = z[k]

        end
    end
```

Exercise

Think of what might be appropriate prior parameters for the Inverse-Wishart distribution. Should its mean be high or low here?

The variational inference procedure for estimating states and the process noise covariance matrix simultaneously requires a bit more thought, but is still very straightforward:

In [8]:
```
# Iterations of variational inference
num_iters = 100

# Initialize variational marginal distributions
init = @initialization begin
    q(z) = MvNormalMeanCovariance(zeros(2), diageye(2))
    q(Q) = InverseWishart(10, diageye(2))
end

# Define variational distribution factorization
constraints = @constraints begin
    q(z_0,z,Q) = q(z_0, z)q(Q)
end

# Variational inference procedure
results = infer(
    model          = LGDS_Q(A=A,C=C, σ=σ, T=T),
    data           = (y = [observations[k] for k in 1:T],),
    constraints    = constraints,
    iterations     = num_iters,
    options        = (limit_stack_depth = 100,),
    initialization = init,
    free_energy    = true,
    showprogress   = true,
)
```

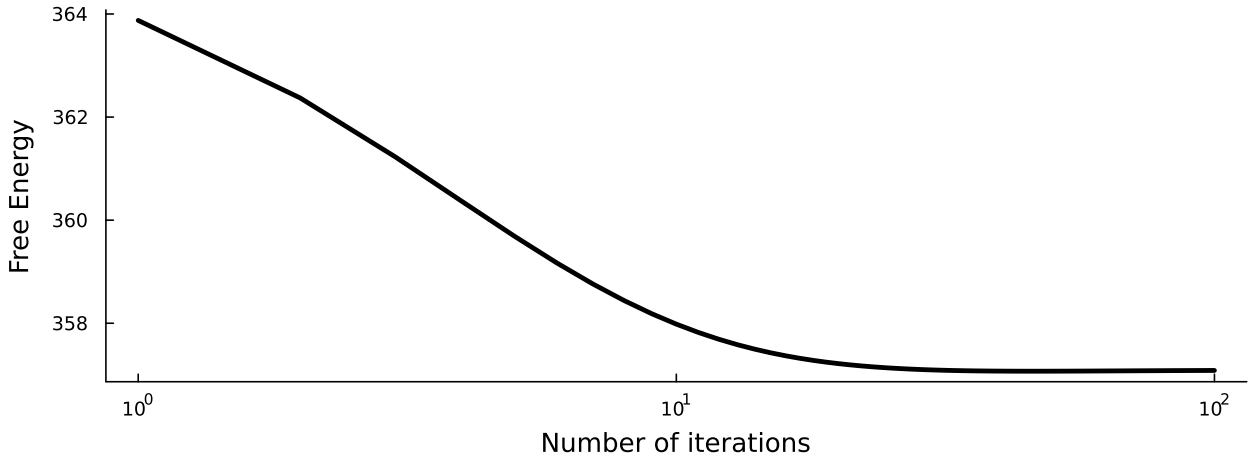
```
Progress: 100%|████████████████████████████████████| Time: 0:00:02
```

Out[8]:
```
Inference results:
  Posteriors       | available for (z_0, Q, z)
  Free Energy:     | Real[363.876, 362.371, 361.232, 360.361, 359.69, 359.169, 358.761, 358.44, 358.185, 35
7.983  …  357.083, 357.083, 357.083, 357.083, 357.083, 357.084, 357.084, 357.084, 357.084, 357.084]
```

Again, let's inspect the free energy to see if we have converged.

In [9]:
```
plot(1:num_iters,
     results.free_energy,
     color="black",
     xscale=:log10,
     xlabel="Number of iterations",
     ylabel="Free Energy",
     size=(800,300))
```

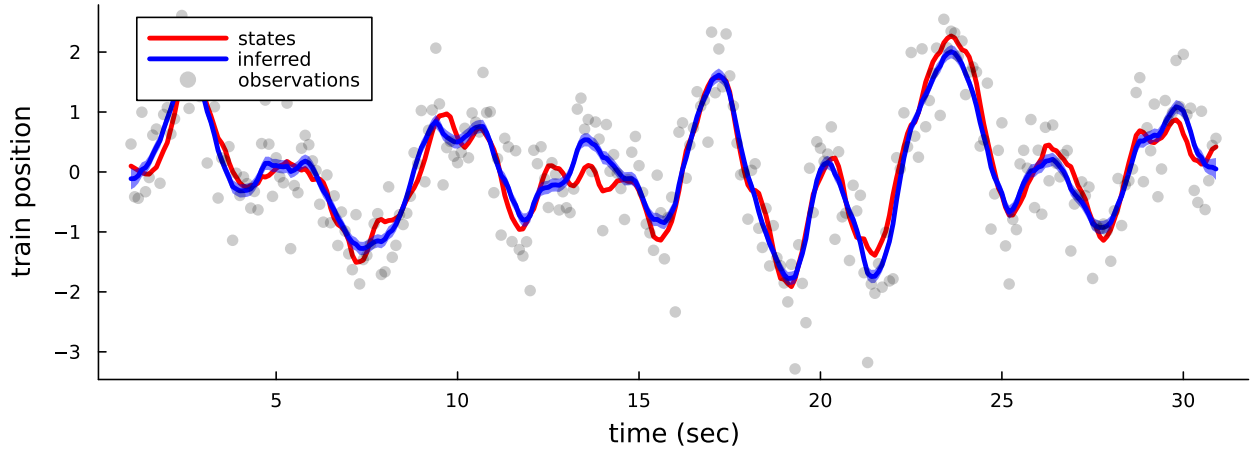
Out[9]:



Alright. That looks good. Let's extract the inferred states and visualize.

In [10]:
```
m_z = cat(mean.(last(results.posteriors[:z]))...,dims=2)
v_z = cat(var.(last(results.posteriors[:z]))...,dims=2)

plot(time, states[1,:], color="red", label="states", xlabel="time (sec)", ylabel="train position")
plot!(time, m_z[1,:], color="blue", ribbon=v_z[1,:], label="inferred")
scatter!(time, observations, color="black", alpha=0.2, label="observations", legend=:topleft, size=(800,300
```

Out[10]:



That's much smoother. The free energy of this model ($\mathcal{F} = 357.08$) is smaller than that of the earlier model with Q set to an identity matrix ($\mathcal{F} = 434.10$). That means that the added cost of inferring the matrix Q is offset by the increase in performance it provides.

The error with respect to the true states seems smaller as well, but in practice we of course can't check this.

Let's inspect the inferred process noise covariance matrix:

In [11]:
```
Q_MAP = mean(last(results.posteriors[:Q]))
```

Out[11]:
```
2×2 Matrix{Float64}:
 0.0600162   0.00526415
 0.00526415  0.146396
```

We do not have enough data to recover the true process noise covariance matrix exactly, but the result is definitely closer to the truth.

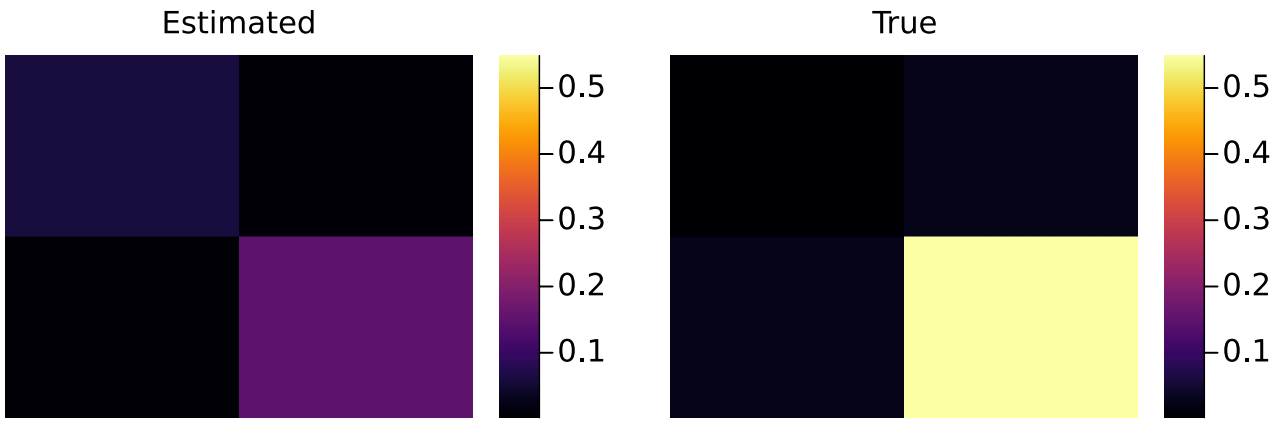
In [12]:
```
# True data
Q_true = data["Q"]
```

Out[12]:
```
2×2 Matrix{Float64}:
 0.00173611  0.0240885
 0.0240885   0.549072
```

Let's visualize that for a closer look:

In [13]:
```
# Colorbar limits
clims = (minimum([Q_MAP[:]; Q_true[:]]), maximum([Q_MAP[:]; Q_true[:]]))

# Plot covariance matrices as heatmaps
p401 = heatmap(Q_MAP, axis=([], false), yflip=true, title="Estimated", clims=clims)
p402 = heatmap(Q_true, axis=([], false), yflip=true, title="True", clims=clims)
plot(p401,p402, layout=(1,2), size=(900,300))
```

Out[13]:



Exercise

Can you come up with a way to improve the model even further?
